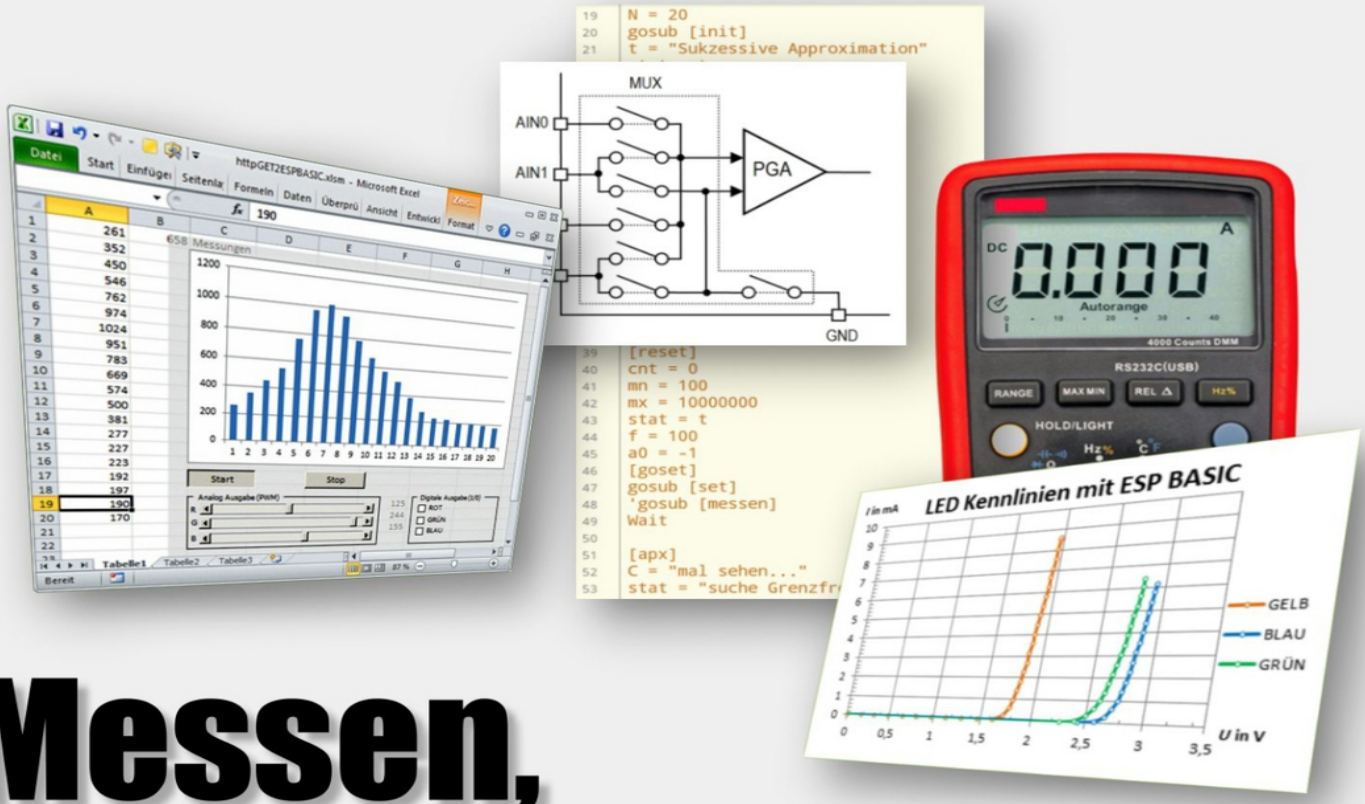
H.-J. Berndt



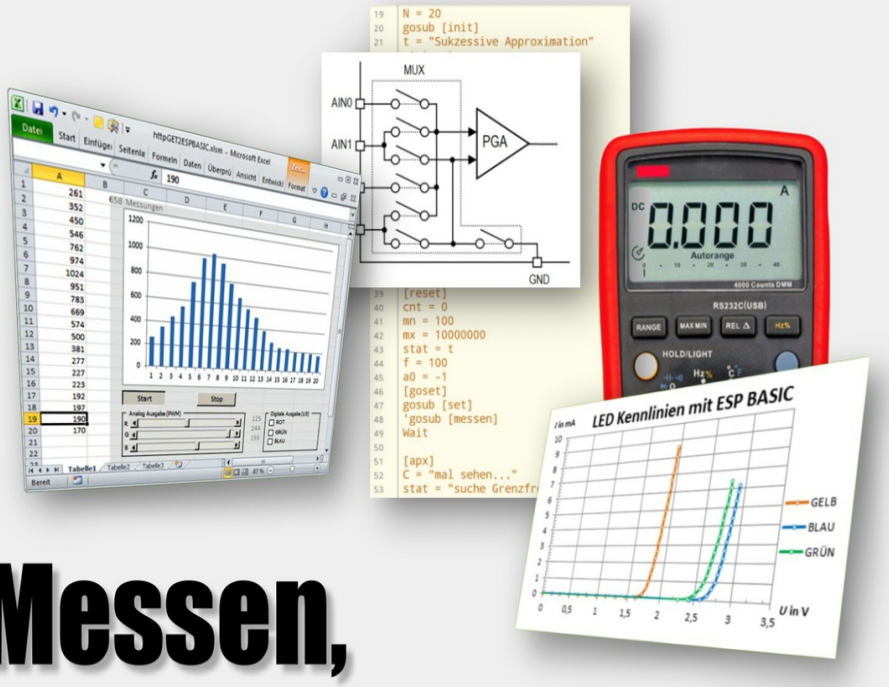
Messen, Steuern und Regeln mit WiFi und ESP 8266 BASIC

Einführung Beispiele Anwendungen

Download
ESP BASIC
esp8266basic.com

Programme und Messdaten im Browser und mehr - drahtlos

H.-J. Berndt



Messen, Steuern und Regeln mit WiFi und ESP8266 BASIC

Einführung Beispiele Anwendungen

Download
ESP BASIC
esp8266basic.com

Programme und Messdaten im Browser und mehr - drahtlos

Hans-Joachim Berndt

Messen, Steuern und Regeln mit WiFi und ESP-BASIC

–

Einführung - Beispiele - Anwendungen

Vorwort

Mess-, Steuerungs- und Regelungstechnik hat sich in den letzten Jahren von der Drahtgebundenheit befreit. Dank preiswerter Hardware und schnellem und überall verfügbarem Internet gewinnt die Messdatenübertragung über WiFi nicht nur in der Industrie, sondern auch im Ausbildungs- und Hobbybereich immer mehr an Bedeutung.

Mit dem ESP8266 kam ein Baustein auf den Markt, der serielle Daten über WiFi weiterleiten kann und umgekehrt. Inzwischen findet man diesen Baustein in Schaltsteckdosen und anderen Gegenständen des täglichen Bedarfs, dem sogenannten Internet of Things IoT. Der Programmierer Michael Molinari (aka mmiscool) schuf einen freien und offenen BASIC-Interpreter für den ESP8266 unter dem Namen ESP8266BASIC, der einmalig in den ESP8266 übertragen werden muss, um dann über WiFi im beliebigen Browser mit dem Interpreter zu kommunizieren. Eigene Programme werden im Flashspeicher des ESP abgelegt und können als Autostart konfiguriert sein.

Dieses Buch möchte die Schwelle zum Einstieg in die Welt von IoT so herab setzen, dass mit geringstem Aufwand und einer Hand voll Programmzeilen eigene Ziele erreichbar sind. Das kürzeste Programm besteht aus einer Zeile und gibt die Temperatur eines Sensors aus. Diese Sprache eignet sich aber auch besonders zur Überprüfung der Machbarkeit von Konzepten, die sich mit Compilersprachen viel zeitaufwändiger und umständlicher gestalten würden. Auf diese Art konnten z. B. ein Multimeter und andere Interfaces dazu bewegt werden ihre Messdaten anstatt über RS232/USB mittels WiFi drahtlos an Excel weiter zu reichen.

Das Buch gliedert sich in vier Abschnitte, wobei am Anfang eine allgemeine Einführung steht, gefolgt von Spezialanweisungen für den ESP8266 und weitere unterstützte Hardware. Anhand von vielen kurzen Beispielen werden die Aufrufe und Befehle aus der Referenz verdeutlicht. Der letzte Abschnitt zeigt komplexere Anwendungen aus dem Bereich

der Mess-, Steuerungs- und Regelungstechnik und greift dabei auf den ersten Teil zurück.

Es wurde meist ein Chrome-Browser auf verschiedenen mobilen Geräten benutzt. Die verwendete Hardware war ein oder mehrere ESP8266 „Witty-Cloud“-Boards, sowie Wemos D1 Mini Versionen. Als ESP8266-BASIC kam überwiegend Version 3, Branch 69 in in der 2 MB-Version, zum Einsatz. Für den ESP32 gibt es zur Zeit dieser Niederschrift kein ESPBASIC.

www.hjberndt.de

November 2019

Inhalt

Literaturverzeichnis

1. ESPBASIC ALLGEMEIN

ESPBASIC verfügt über alle grundlegenden Eigenschaften und Befehle anderer BASIC-Dialekte. Wer schon einmal mit BASIC in Berührung kam, kann hier ohne hohen Lernaufwand loslegen und möglicherweise sofort zu Kapitel 2 springen, um die speziellen Befehle kennen zu lernen.

BASIC ist ein Interpreter. Das bedeutet, dass jede geschriebene Programmzeile analysiert und interpretiert wird und bei fehlerfreier Syntax bzw. Grammatik sofort ausgeführt wird. Im Gegensatz dazu gibt es Hoch-Sprachen, wie C/C++, Java und Pascal, die erst übersetzt bzw. kompiliert - und dann in den jeweiligen Maschinenkode übertragen werden. Manche BASIC-Varianten ermöglichen es Programme in Maschinensprache umzuwandeln, für ESPBASIC ist dieser Weg nicht vorgesehen.

Die Sprache BASIC verfügt im Allgemeinen über Sprachelemente, die auch in ESPBASIC vorhanden sind. Diese gemeinsamen Elemente sollen im nachfolgenden ersten Abschnitt anhand von ESP-Beispielen erläutert und getestet werden. Dazu ist es erforderlich die im Anhang angegebene Installation von ESPBASIC einmalig auf einem ESP8266 durchzuführen. Die Programmierung erfolgt dann im Browser auf Hardware eigener Wahl. Hier wird ein ESP8266 mit seinem BASIC über einen Router mit der vergebenen IP 192.168.178.39 aufgerufen und angesprochen. Im Browser-Editor erfolgt die Programmierung.

1.1 EIN- UND AUSGABEN

Fast alle klassischen BASIC-Varianten verfügen über die beiden Grundbefehle zur Ein- und Ausgabe. „Hallo Welt“ als erster Satz einer „neuen“ Programmiersprache braucht immer einen Ausgabekanal. Darum erfolgt die erste Kontaktaufnahme mit der Außenwelt oft mit diesen Worten. Der Ausgabekanal ist hier zunächst der Browser. Weiter unten

wird gezeigt, wie auch andere Anzeigen und Kanäle zur Anwendung kommen können.

1.1.1 PRINT

Das erste Programm lautet also – wie in fast jeder BASIC-Variante:

Print “Hallo Welt”

Im Browser sieht das dann etwa so aus:

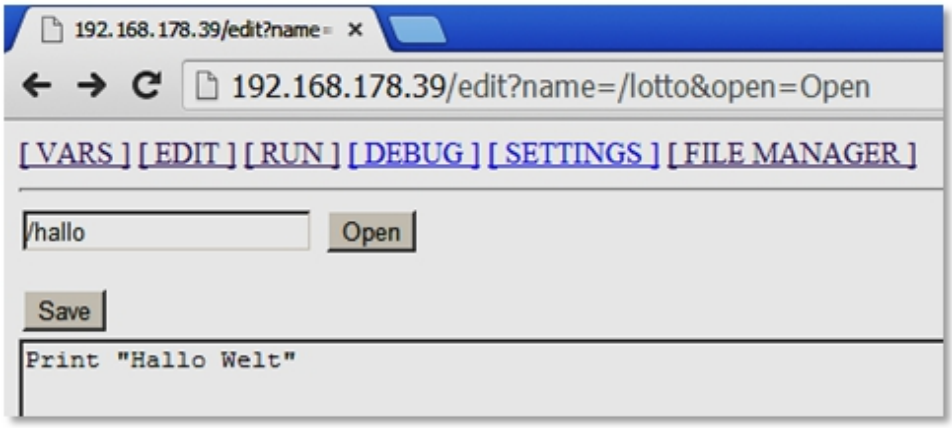


Abbildung 1-1: Erste Ausgabe

Nach „SAVE“ startet das Programm mit „RUN“. Das Ergebnis erscheint ebenfalls im Browser:

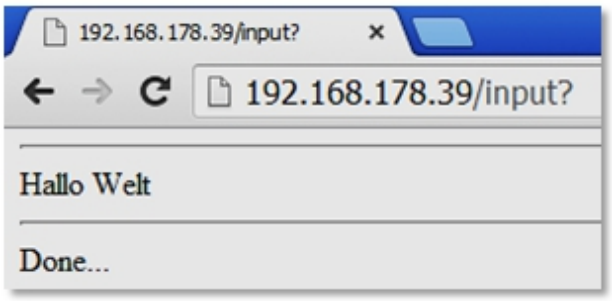


Abbildung 1-2: Erste Schritte

ESPBASIC liefert die Ausgabe mit diesem Kommando allerdings zusätzlich auch noch über die serielle Schnittstelle, also am Anschluss TX des ESP8266. Ein entsprechendes USB-Interface würde diese Meldung ebenfalls empfangen. Soll nur der eine oder andere Kanal bedient werden, stellt ESPBASIC die Befehle html oder wprint und serialprint zur Verfügung. Beispiele dazu befinden sich an entsprechender Stelle weiter unten im Abschnitt ESPBASIC Speziell.

1.1.2 INPUT

Der übliche BASIC-Eingabebefehl ist vorhanden, entspricht hier jedoch einem *serialinput* , benutzt also die serielle Leitung RX der ESP-Hardware. Ist ein Phone per USB-Adapter/App angeschlossen, könnte diese Variante funktionieren. Achtung: *input* wartet bis eine serielle Eingabe erfolgt. Der ESP ist bis dahin nicht mehr über den Browser erreichbar. *Serialinput* ist ebenfalls im Abschnitt 3 *ESPBASIC Speziell* aufgeführt und erläutert.

1.1.3 INPUT IM BROWSER

Soll die Eingabe auf dem Bildschirm, also im Browser, erfolgen, so steht aus dem Bereich WEB-Interface eine *textbox* zur Verfügung, die hier schon einmal kurz herangezogen werden soll. Auch ein *branch* und ein *button* kommen jetzt schon kurz zum Einsatz.

```
textbox a
button “OK”, [mach]
wait
[mach]
print a
```

Es wird eine Eingabebox dargestellt und der Inhalt mit der Variablen *a* verknüpft. Ein *button* mit dem Schriftzug „OK“ sorgt dafür, dass bei Betätigung das Programm zur Zeile mit dem Label [*mach*] springt. Dort wird der Inhalt von *a* mit *print* kontrolliert.

Diese Variante erscheint möglicherweise aufwändiger als das schlichte *INPUT* , doch die Möglichkeiten und der Komfort lassen sich vielleicht schon erahnen.

1.2 SCHLEIFEN UND WIEDERHOLUNGEN

Hochsprachen benutzen in der Regel drei Schleifenarten bzw. Wiederholstrukturen. Mit diesen Anweisungen können

mehrfach gleiche Dinge erledigt werden. Soll z. B. dreimal „Hallo Welt“ nach einander ausgegeben werden, so könnte das wie folgt programmiert werden:

```
Print “Hallo Welt”
Print “Hallo Welt”
Print “Hallo Welt”
```

Abgesehen von der Schreibarbeit bzw. Kopier- und Einfüge-Arbeit ist diese Art der Programmierung speicherintensiv. Stil wäre ein weiteres Stichwort.

1.2.1 FOR NEXT

Mit der sogenannten *For/Next* Schleife kann die Anzahl einfach abgezählt werden. Damit ist es kein Problem mehr die Meldung auch 10000-mal auszugeben.

```
For i = 1 to 4
 Print i & ” Hallo Welt”
Next i
```

Das Ergebnis sind vier Zeilen mit dem gewünschten Text. Diese Wiederholung eignet sich insbesondere, wenn die genaue Anzahl der Wiederholungen bekannt ist. Die Erhöhung des Zählers erfolgt ohne weitere Angaben um genau 1. Mit der Schrittangabe *step* kann darauf Einfluss genommen werden. Mit der Schrittgröße 0.5 verdoppelt sich die Zahl der Durchläufe.

```
for i = 1 to 4 step 0.5
 print “Hallo Welt”
next i
```

Eine Rückwärtszählung, die bei dieser Ausgabe wenig Sinn macht, aber an anderer Stelle durchaus hilfreich sein kann, erfolgt durch Angabe einer negativen Schrittweite. Der Countdown der Ausgabe:

```
for i = 4 to 1 step -0.5
 print “Hallo Welt”
next i
```

Bei Ausgaben von Quasi-Analog-Anschlüssen über PWM lassen sich solche Schleifen komfortabel einsetzen. Einfache

Verzögerungen sind damit auch möglich. Ein *delay* -Befehl ist jedoch ebenfalls vorhanden.

1.2.2 Do Loop

Soll eine Wiederholung so lange erfolgen, bis eine bestimmte Bedingung erfüllt ist, bieten sich andere Strukturen an. In einigen Sprachen sind diese bekannt als While/Wend und Repeat/Until; übersetzt: Solange und Wiederhole/Bis. ESPBASIC kennt nur die Do/Loop Struktur, die sich wie eine Repeat/Until-Schleife verhält. Die Bedingung wird am Ende, also nach einmaligem Durchlauf überprüft. Tritt eine Bedingung nie ein, entsteht eine Endlos-Schleife. In ESPBASIC programmierte Endlosschleifen ohne Verzweigungen führen dazu, dass BASIC über den Browser nicht mehr erreichbar ist. Ein Reset kann erforderlich werden.

Um drei Wiederholungen zu erhalten, liefert folgender Kode das gewünschte Resultat:

```
i = 1
do
 Print “Hallo Welt”
 i = i + 1
loop until i > 3
```

Der Anfangswert wird auf 1 gesetzt und anschließend die Schleife betreten. Nach der Ausgabe erfolgt die Erhöhung um 1, wonach die Bedingung überprüft wird, ob der Wert schon größer als 3 ist. Wenn diese Bedingung wahr ist, endet die Schleife.

ESPBASIC erlaubt noch ein *do/loop while* , was aber lediglich auf eine invertierte Bedingung hinaus läuft und leider mit einer abweisenden *while* -Struktur nichts gemeinsam zu haben scheint.

Hallo Welt

Hallo Welt

Hallo Welt

Done…

1.3 VARIABLEN

Eine Variable ist ein Speicherplatz für einen Wert, der mit einem Namen angesprochen werden kann. ESPBASIC unterscheidet zwischen numerischen- und Zeichenketten-Variablen. Bei Variablennamen wird zwischen Groß- und Kleinschreibung unterschieden.

1.3.1 ZAHLEN

Im obigen Schleifen-Beispiel wurde die Variable mit dem Namen *i* benutzt, um einen sich ändernden Zahlenwert zu speichern. Dort wurde diese Variable bei jedem Durchlauf der Wiederholung um den Wert 1 erhöht. Dies ist eine der primitivsten Operationen mit numerischen Variablen bzw. Zahlen. Weitere Grundoperationen mit numerischen Variablen sind Subtraktion (-), Multiplikation (*), Division (/), Potenzieren (^) und Modulo (%). Ein Beispiel mit Ausgabe:

Zahl = 88

Print Zahl^3

681472

Done…

Print 3 - 2

Print 3 * 2

Print 3 / 2

Print 3 % 2

Print 3^2

1

6

1.5

1

Done…

Numerische Funktionen in Abschnitt 1.5 können weitere Rechnungen durchführen.

1.3.2 Zeichenketten (Strings)

Die zweite Variablenart in BASIC ist die Zeichenkette (String) und wird oft mit dem $-Zeichen gekennzeichnet. Zeichenketten-Konstanten werden in Anführungszeichen geschrieben. Die Zeichenfolge „Hallo Welt“ ist eine Zeichenkette mit 10 Zeichen. Sie kann als Variable abgespeichert werden und entsprechend mit Zeichenkettenfunktionen untersucht oder verändert werden. Inhalte von Internetseiten sind meist auch Zeichenketten. Als Grundoperator für Zeichenketten verfügt auch ESPBASIC über die Addition (&), die allerdings keine Zahlenwerte addiert, sondern Zeichenketten verbindet.

```
A$ = “Hallo Welt”
Print A$
A$ = A$ & “!”
Print A$
```

Hallo Welt

Hallo Welt!

Done…

Die Zeichenkettenvariable *A$* (gesprochen: A-String) bekommt den Wert der Konstanten „Hallo Welt“ zugewiesen. Der Inhalt der Variablen wird ausgegeben und in der nächsten Zeile eine konstante Zeichenkette mit als Inhalt ein Ausrufezeichen angehängt und schließlich ausgegeben.

1.3.3 Arrays

Eine Ansammlung gleicher Variablentypen, die indiziert angesprochen werden, nennt BASIC ein Array. Sollen zum Beispiel Messwerte gespeichert und verarbeitet werden, bieten sich solche Speicher an. Sie sind vergleichbar mit einem

Taschenrechner mit mehreren STORE-Möglichkeiten. Zur Verdeutlichung sollen zehn X-Werte und zehn Y-Werte berechnet und ausgegeben werden. In den X-Speichern von 1 bis 10 sollen auch die Zahlen 1 bis 10 gespeichert werden. Die zehn Y-Werte erhalten das jeweilige Quadrat der X-Werte, also Zahlen von 1 bis 100. In BASIC werden Arrays dimensioniert mit *DIM* :

```
DIM x(10)
DIM y(10)
‘Berechnung
For i = 1 to 10
 x(i) = i
 y(i) = i * i
Next i
‘Ausgabe
For i = 1 to 10
  Print x(i) & ” ….. ” & y(i)
Next i
```

---

1 ….. 1

---

2 ….. 4

---

3 ….. 9

---

4 ….. 16

---

5 ….. 25

---

6 ….. 36

---

7 ….. 49

---

8 ….. 64

---

9 ….. 81

---

10 ….. 100

Done…

Das nächste Beispiel verwendet ein Array für fünf Zeichenketten. Die fünf Zeichenketten *A$* erhalten mit Hilfe einer weiter unten aufgeführten *str* -Funktion einen numerischen Wert der Indexvariablen *i* als Zeichenkette gewandelt zugewiesen. Zusätzlich wird noch eine Konstante angehängt und im Array abgelegt. Nach der Ausgabe wird mit *undim* aufgeräumt und der Speicherplatz freigegeben.

```
DIM a$(5)
For i = 1 to 5
  a$(i) = str(i) & ". Eintrag"
Next i
'Ausgabe
For i = 1 to 5
 Print a$(i)
Next i
UNDIM a$
```

1. Eintrag

2. Eintrag

3. Eintrag

4. Eintrag

5. Eintrag

Done…

1.4 OPERATOREN

Neben den Grundoperationen, die bei den Variablen weiter oben angegeben sind, gibt es auch in diesem BASIC weitere Operatoren: Vergleichsoperatoren, logische, sowie boolesche Operatoren.

1.4.1 VERGLEICHSOPERATOREN

Das Ergebnis eines Vergleichs ist in der Regel ein boolescher Ausdruck oder auch Wahr oder Unwahr; true oder false bzw.

in BASIC 0 und -1. Dabei entspricht der boolesche Zustand Unwahr einer 0 (numerisch) und Wahr eine -1 (numerisch). Alle Vergleichsoperatoren funktionieren mit Variablen oder Konstanten. Hier ein Test für Vergleiche mit Ausgabe:

Print 5 = 5

Print 5 = 6

Print 1 < 2

Print 1 > 2

Print 1 <> 2

Print 1 >= 2

Print 2 <= 2

Print 0.0 = 0

Print “Hallo” = “hallo”

Print “Hallo” = “Hallo”

-1

0

-1

0

-1

0

-1

-1

0

-1

Done…

Nun könnte man mit diesen Ergebnissen rechnen, da es ja numerische Werte sind. Eine Probe wäre: Print (5 = 5) + (5 = 6) + (1 < 2) + (1 > 2) + (1 <> 2) + (1 >= 2) + (2 <= 2). Ohne Klammern macht BASIC Schwierigkeiten, mit Klammern wird die Zeile interpretiert und das Ergebnis zu -4 berechnet.

Man könnte auch multiplizieren oder dividieren. Übrigens bei einer Division durch Null liefert dieses BASIC „inf“ (unendlich), als nicht-numerischen Wert. Eine entsprechende Spielerei liefert die Zeile: Print (1/0) & “ormatik”.

1.4.2 Logische Operatoren

Logische Operatoren kommen zum Einsatz, wenn mehrere boolesche Ergebnisse vorliegen. Die Frage könnte lauten: Ist *a = 10* und *b = 40* ? Das UND ist hier ein logisches UND. Das Ergebnis ist Wahr oder Unwahr, also hier -1 oder 0. Eine Überprüfung wäre:

a = 10

b = 40

Print a = 10 and b = 40

Das Ergebnis ist -1, also *Wahr* . Es müssen beide Bedingungen *Wahr* sein damit ein *Wahr* erreicht wird. Wird eine der vier konstanten Zahlen geändert, ergibt das 0 (Unwahr). Hinweis: Groß- und Kleinschreibung der Variablen sollte einheitlich sein.

Die logischen Operatoren sind *AND* , *OR* , *NOT* , *XOR* ; also UND, ODER, NICHT und exklusiv ODER. Alle Regeln der booleschen Algebra könnte man hier überprüfen, oder einfach nur ein ODER-Gatter mit zwei Eingängen und Wahrheitstabelle programmieren.

Bei einem ODER mit zwei Eingängen gibt es 4 Möglichkeiten.

Print 0 or 0

Print 0 or 1

Print 1 or 0

Print 1 or 1

0
1
1
1

Done…

So lässt sich durch eine kleine Änderung auch das XOR überprüfen.

1.4.3 Boolesche Operatoren

Boolesche Operatoren lassen sich gut bei Bitmanipulationen einsetzen, wie sie bei digitalen Ein- und Ausgaben vorkommen. ESPBASIC gibt *AND* , *OR* , *XOR* , *NOT* , << und >> an. Ein bitweises UND liefert bei Print 1 AND 3 eine 1. Das wird plausibel, wenn man sich die Zahlen binär vorstellt. In 4-Bit-Schreibweise wäre das

0001 für die 1

0011 für die 3

Da bei einer UND-Verknüpfung alle Eingänge 1 sein müssen, um eine 1 zu erhalten, ist dies nur an der niedrigsten Stelle (rechts) beider Zahlen der Fall, welche einer 1 entspricht. Mit der UND-Verknüpfung lassen sich also bestimmte Bits überprüfen. Soll in einem Byte das höchste Bit überprüft werden, so wird es mit 128 UNDiert. Das Ergebnis ist 0 oder 128, je nach Zustand vom Bit 2^7 (128). Die entsprechenden Zeilen in ESPBASIC lauten:

```
x = 255
print x and 128
print x and 2^7
```

Alle Werte für *x* unter 128 liefern 0. Beide Print-Varianten sind gleichwertig.

Ähnlich verhält es sich mit den anderen logischen Operatoren. Soll ein bestimmtes Bit „umgedreht“ werden (toggle), kann das mit *XOR* realisiert werden. Angenommen Bit 7 des Bytes soll immer wieder wechseln, weil eine dort angeschlossene Leuchtdiode blinken soll, so könnte das in ESPBASIC wie folgt gelöst werden:

```
x = 255
print x
x = x xor 128
print x
```

x = x xor 128

print x

255

127

255

In binärer Schreibweise:

11111111

01111111

11111111

Mit der binären Schreibweise werden auch die beiden Schiebeoperatoren deutlich. Schiebt man ein Bit um eine Stelle nach links, so verdoppelt sich die Wertigkeit um eine Zweierpotenz. Aus einer 4 (2^2) wird eine 8 (2^3), anders ausgedrückt verdoppelt sich der Wert; aus einer 3 wird eine 6.

0011 – 3 << 1

0110 – 6

Eine 12 erhält man demnach mit 3 << 2. Und in der Tat liefert *print 3 << 2* eine 12.

Der umgekehrte Weg ist die Halbierung bzw. die Division durch 2. Mit *print 6 >> 1* sollte eine 3 erscheinen. Wegen der Ganzzahldivision ergibt 6 >>2 eine 1, das Bit 2^0 bleibt übrig.

Mit diesen Operationen können einzelne Bits isoliert und binär an eine bestimmte Stelle geschoben werden.

Um im 8-Bit-Bereich eines Bytes zu bleiben ist es sinnvoll numerische Werte in BASIC mit 255 zu UNDieren. Beispielsweise zeigt BASIC bei Print NOT (1) ein -1, was ja logisch erscheint. Dies ist jedoch keine 254, die binär erwünscht wäre. Sollen jedoch die 8 Bits eines Bytes negiert werden, so führt folgende Vorgehensweise zum Ziel: *Print NOT (1) AND 255* . Dies liefert die Negation von 00000001, also 11111110. Eine weitere Probe mit dem Bitmuster 10101010 und der Negation 01010101. Die Zahl 128+32+8+2 sind die gesetzten Ausgangsbits. In BASIC liefert die Zeile

Print NOT (128+32+8+2) AND 255 den Wert 85, was die Summe von 64+16+4+1 ist.

Mit den nun folgenden Zeichenketten-Funktionen lassen sich Dezimalzahlen in Binärschreibweise wandeln. Entsprechende Beispiele sind dort gelistet.

1.5 FUNKTIONEN

Funktionen liefern Werte oder führen Berechnungen und Umwandlungen durch. Vielen Funktionen kann ein Wert oder eine (oder mehrere) Variable(n) übergeben werden, womit die Funktion dann arbeitet und das gewünschte Ergebnis liefert. Wie aus der Überschrift schon zu erkennen ist, gibt es numerische Funktionen und solche die mit einzelnen Zeichnen oder ganzen Zeichenketten umgehen können. Auch die Zeit wird über Funktionen abgerufen.

1.5.1 MATHEMATISCHE UND NUMERISCHE FUNKTIONEN

Diese Funktionen arbeiten mit Zahlen und lassen sich in verschiedene Abschnitte gliedern.

Mathematik

Die mathematischen Funktionen in ESPBASIC sollen den Anfang machen. Es sind dies:

Quadratwurzel	*sqr()*
Potenz	*pow()*
Logarithmus	*log()*
Exponentialfunktion	*exp()*
Sinus	*sin()*
Cosinus	*cos()*
Tangens	*tan()*

Arcus Sinus	*asin()*
Arcus Cosinus	*acos()*
Arcus Tangens	*atan()*

Das Quadrieren wurde weiter oben bei den Arrays bereits benutzt. Der Ausdruck x^2 kann in BASIC auf verschiedene Arten formuliert werden. Möchte man 64 als Quadrat der Zahl 8 berechnen, so liefern die folgenden Ausdrücke das Ergebnis:

```
Print 8 * 8
Print 8^2
Print pow(8,2)
```

Dies funktioniert auch mit Variablen:

```
x = 8
y = 2
Print x * x
Print x^y
Print pow(x,y)
```

Eine Umkehrfunktion ist die Quadratwurzel, die mit *sqr* (SquareRoot) berechnet wird.

```
Print sqr(8*8)
```

Dieser Ausdruck liefert wieder die Zahl 8. Spielt man noch etwas mit der Zahl 8 und ändert die Zahl 2 in die natürliche Zahl *e* , kann folgendes Beispiel die verschiedenen mathematischen Funktionen überprüfen:

```
y = 8
x = exp(1)
Print x
Print x^y
x = pow(x,y)
Print log(x)
```

Mit der Exponentialfunktion und 1 als Übergabeparameter, also e^1 , kommt die natürliche Zahl $e = 2.71828$ selbst zum Vorschein, da jede Zahl mit der Potenz 1, die Zahl selbst ist.

Anschließend folgt „x hoch y“ und liefert 2980.94263. Die nächste Zeile berechnet dieses Ergebnis auf die andere Art und kopiert den Wert in die Variable *x* . Nun folgt zum Schluss die Umkehrfunktion der *e* -Funktion – der Logarithmus, wodurch wieder die Zahl 8 berechnet wird. Aufgrund der Fließpunkt-Arithmetik schreiben BASIC und auch andere Sprachen 7.99999.

Die *log()* -Funktion ist hier also der natürliche Logarithmus, der oft auch mit *ln()* oder *lg()* in anderen Sprachen aufgerufen werden kann. *Log()* ist dabei oft der Zehner-Logarithmus, also der Basis 10. Da zwischen den verschiedenen Logarithmen ein konstanter Bezug besteht, erhält man den Zehnerlogarithmus aus dem Ausdruck *log(x)/log(10)* . Wird die Zahl der Nullen der Zahl 1000 (also 10^3) gesucht, so klappt das in BASIC wie folgt: Print log(1000)/log(10), oder

```
x = 1000
lg = log(x)/log(10)
Print lg
```

Sollen Messwerte in einem logarithmischen Maßstab dargestellt werden, erweist sich diese Funktion auch in BASIC als nützlich.

Mit sechs Winkelfunktionen deckt ESPBASIC diesen Bereich ab. Wie üblich erwarten und liefern diese Funktion die Winkel in Bogenmaß, also in Vielfachen von *Pi* . Dabei entspricht der Kreisumfang von 360° 2*Pi* . Soll der Winkel in Grad übergeben werden, so muss eine Umrechnung erfolgen. Dazu benötigt man die Zahl *Pi* selber. In ESPBASIC ist eine solche Konstante nicht deutlich dokumentiert bzw. nicht vorhanden. Da der Cosinus von 180 Grad aber -1 ergibt, benutzt man einfach die Umkehrfunktion Arcus Cosinus, die ja den zugehörigen Winkel zu einem Cosinus Wert liefert mit der Übergabe -1, also *pi = acos(-1)* .

```
pi = acos(-1)
Print pi
winkel = 90
Print sin(winkel/180*pi)
```

Die erste Ausgabe ist die Zahl 3.14159. Danach wird ein Winkel von 90 Grad festgelegt und die Sinusfunktion mit der Umrechnung aufgerufen, die die Zahl 1 als Ergebnis liefert. Als letztes Beispiel soll noch herausgefunden werden bei welchem Winkel eine Steigung 1 oder 100% beträgt. Da der Arcus Tangens den Winkel zwischen Gegenkathete zu Ankathete liefert, sieht der Aufruf in BASIC wie folgt aus:

```
pi = acos(-1)
Print pi
winkelbogen = atan(1)
winkel = winkelbogen/pi*180
Print winkel
```

Nach der Nachfrage der Zahl *Pi* wird das Ergebnis in Bogenmaß abgerufen und anschließend wieder in einen Winkel in Grad umgewandelt, der dann mit 45 Grad berechnet wird. Eine Abweichung liegt an den verschiedenen Zuweisungen und vermutlich internen Rundungen. Die Zeile *Print atan(1)/acos(-1)*180* liefert genau 45.

Sollen Servomotoren zum Einsatz kommen (Abschnitt 2), könnten sich diese Funktionen als nützlich erweisen.

Weitere mathematische Funktionen

Absolut Funktion Ganzzahl	*abs()*
Absolut Funktion Fließpunkt	*fabs()*
Abrundungsfunktion auf Ganzzahl	*floor()*
Aufrundungsfunktion auf Ganzzahl	*ceil()*
Rundung auf Ganzzahl	*round()*

```
x = log(0.7)
print x
print abs(x)
print fabs(x)
```

Die drei Print-Ausgaben liefern: -0.35667, 0, 0.35667. Erweitert mit den Rundungsfunktionen:

```
x = -30.50
print x
print abs(x)
print fabs(x)
print floor(x)
print ceil(x)
print round(x)
```

```
-30.5
30
30.5
-31
-31
-31
```

Done…

Verschiedene numerische Funktionen

Eine Werkzeugkiste voller Nützlichkeiten liefert diese BASIC-Variante mit folgenden Aufrufen:

Zufallszahl	*rnd()*
Systemzeit	*millis()*
Zeichenkette zu Ganzzahl	*int()*
Zeichenkette zu Zahl	*val()*
Oktal Zahl	*oct()*
Hexadezimalzahl	*hex()*

Hexadezimal zu Dezimal	*hextoint()*
Freier Speicher	*ramfree()*
Freier Datei-Speicher	*flashfree()*
BASIC-Version	*version()*
Chip-ID	*id()*

Sechs Zufallszahlen aus dem Bereich 1 bis 49 erhält man in BASIC mit diesem Dreizeiler:

```
for i = 1 to 6
 print 1 + rnd(49)
next i
```

Dabei findet keinerlei Sortierung oder Überprüfung auf doppeltes Vorkommen statt.

Um berechnete Messwerte etwas schwanken zu lassen, um eine höhere Realitätsnähe durch zufällige Fehler zu realisieren, kann ein konstanter theoretischer Temperaturwert, der um ein Grad nach oben oder unten abweichen kann (Delta T = 2000 mK), wie folgt mit drei Nachkommastellen berechnet werden:

```
dt = 2000
mt = dt/2/1000
for i = 1 to 20
 r = rnd(dt)/1000
 print 40.0 + (mt-r)
next i
```

40.122

40.829

39.536

40.243

40.446

39.153

40.211

40.633

39.34

40.047

39.106

39.813

40.234

40.941

40

40.422

39.129

39.836

40.609

39.316

Done…

Bei Messungen ist oft auch die Zeit beteiligt. Die Funktion *millis()* liefert die Zeit in Millisekunden, die seit dem Einschalten des ESP verstrichen ist. Eine sehr einfache Zeitschleife erhält man mit Hilfe der Funktion *delay(),* die um eine gewisse Anzahl vom Millisekunden verzögert. Zu Anfang wird der aktuelle Wert von *millis()* erfragt, in *t0* gespeichert und ausgegeben. Danach werden 10 Zeiten ausgegeben mit einer Verzögerung von 0,5 Sekunden. Die Ausgabe erscheint erst nachdem (!) das Programm beendet wurde im Browserfenster.

```
t0 = millis()
print t0
for i = 1 to 10
 print (millis()-t0)
 delay 500
next i
```

Ergebnis:

88686328

8

512

1016

1520

2024

2528

3032

3536

4040

4552

Done…

Weiter unten kommen andere Zeitfunktionen zur Anwendung.

Nach der Systemzeit kann mit *version()* die aktuell installierte ESPBASIC-Version erfragt werden. *Id()* liefert eine Chip-ID des ESP; den freien Systemspeicher liefern die Funktionen *ramfree()* und *flashfree()*. Am Anfang des folgenden Tests wird die Funktion *val()* untersucht, die aus einer Zeichenkette einen numerischen Wert liefert. Interessanterweise bricht die Funktion bei nicht numerischen Zeichen nicht mit einem Fehler ab, sondern liefert den Wert, der bis dort interpretiert wurde. Mit *int()* erhält man den ganzzahligen Anteil einer Fließpunktzahl. Die verfügbaren Speichergrößen werden so in ganzen kB angegeben:

```
a$ = “200 mV”
print val(a$) & ” Ohm”
print int(2.89)
print version()
print int(ramfree()/1024) & ” kB freier Ram-Speicher”
print int(flashfree()/1024) & ” kB freier Flash-Speicher”
print id()
```

200 Ohm

2

ESP Basic 3.0.Alpha 69

11 kB freier Ram-Speicher

905 kB freier Flash-Speicher

12005358

Done…

Beim Umgang mit Bits und Bytes sind Umwandlungen in andere Zahlensysteme von Vorteil. Zwar bietet ESPBASIC keine direkte Binärumwandlung, jedoch werden Oktal- und Hexadezimalzahlen unterstützt. Insbesondere mit Hexadezimalzahlen lassen sich binäre Zustände leicht erfassen. Byte-Darstellungen sind oft in hexadezimaler Form, da Sie z. B. maximal zweistellig werden. Ein sogenannter Hex-Dump stellt Speicherinhalte oft als ASCII-Zeichen und als HEX-Wert dar. Ein Ausrufezeichen „!“ mit dem ASCII-Wert 33 entspricht dann 21_H , da 2*16 + 1 der Dezimalzahl entspricht. Hexadezimalzahlen benutzen die Basis 16. Die einzelnen Zustände werden von 0 bis 9 und weiter mit A bis F bezeichnet. Die Wertigkeit von A bis F ist 10 bis 15. Angenommen ein Byte soll binär 11000011 enthalten. Um schnell die hexadezimale Schreibweise zu erhalten teilt man das Byte in der Mitte in 1100 und 0011. Die linke Hälfte entspricht der Zahl 8 + 4 = 12, in Hex C, die rechte Hälfte stellt eine 2 +1 = 3 dar. Das ist auch hexadezimal eine 3. Zusammengesetzt ergibt das Bitmuster $C3_H$. Wer öfter mit hexadezimalen Bytes umgeht, der „sieht“ quasi die Bits. Der Dezimalwert ist dafür meist wenig hilfreich. Darum sind Umwandlungsroutinen dieser Art überaus hilfreich.

print hextoint(“C3”)

print hextoint(“C”)

print hextoint(“3”)

print hex(195)

print upper(hex(195,4))

195
12
3
c3
00C3

Done…

1.5.2 ZEICHENKETTENFUNKTIONEN

Messwertübertragung und Auswertung sind ohne Zeichenketten kaum vorstellbar. Internetdaten sind oft statische oder generierte Texte im HTML-Format. Der Umgang mit Zeichenketten und Zeichen erfordert Zeichenkettenfunktionen.

Anzahl der Zeichen	*len()*
Zeichensuche	*instr()*
Zeichensuche rückwärts	*instrrev()*
Zeichen innerhalb	*mid()*
Zeichen links	*left()*
Zeichen rechts	*right()*
Zahl zu Zeichen	*str()*
Leerzeichen entfernen	*trim()*
Zeichen ersetzen	*replace()*
Zeichen zu Großbuchstaben	*upper()*
Zeichen zu Kleinbuchstaben	*lower()*

ASCII-Wert Zeichen	*asc()*
Zeichen des ASCII-Werts	*chr()*
Wort einer Zeichenkette	*word()*

Alle Funktionen sollen in einem einzigen Listing mit Print-Ausgaben getestet werden. Neben der Print-Ausgabe ist eine Variablen-Zuweisung ebenfalls möglich. Bei allen Funktionsnamen wird nicht zwischen Groß- und Kleinschreibung unterschieden. Bei Variablennamen wird sehr wohl unterschieden, was hin und wieder zu Fehlern führen kann, da andere BASIC-Dialekte dort nicht unterscheiden. Auch darf zwischen Funktionsname und der ersten Klammer kein Leerzeichen stehen. Der geneigte BASIC-Programmierer könnte die Ergebnisse als kleine Herausforderung voraussagen:

```
a$ = “23:50:06 Donnerstag 200 mV”
print len(a$)
print instr(a$,”:”)
print instrrev(a$,”:”)
print mid(a$,3,3)
print left(a$,8)
print right(a$,2)
print str(22*2)
print replace(a$,“Donnerstag”,“Freitag”)
print trim(“  (Hallo  Welt)  ”)
print upper(a$)
print lower(a$)
print word(a$,3)
print asc(“!”)
print chr(33)
```

26

3

6

:50

23:50:06

---

mV

44

23:50:06 Freitag 200 mV

(Hallo Welt)

23:50:06 DONNERSTAG 200 MV

23:50:06 donnerstag 200 mv

200

33

!

Done…

ESPBASIC kann eine Dezimalzahl in eine Zeichenkette mit hexadezimalen Zeichen wandeln und benutzt dazu *hextoint()* bzw. *hex()*. Soll eine Dezimalzahl in eine Binärzahl gewandelt werden, kann dies in einem kleinen Unterprogramm erledigt werden.

Zuerst von Binär nach Dezimal (bin2dez):

```
b$="10101010"
gosub [bin2dez]
print dez & " = " & b$ & " 0x" & hex(dez)
wait
[bin2dez]
dez=0
n = len(b$)
for bit = n-1 to 0 step -1
 dez=dez+val(mid(b$,n-bit,1))*pow(2,bit)
next bit
return
```

---

170 = 10101010 0xaa

In der anderen Richtung legt die Länge von *b$* fest, wie viele Stellen berechnet werden.

```
dez = 53635
b$="8888888888888888"
gosub [dez2bin]
print b$
wait
[dez2bin]
n = len(b$)
for bit = n - 1 to 0 step -1
 mid(b$,bit+1,1)=chr(48+dez%2)
 dez = dez / 2
next bit
return
```

1101000110000011

Anwendungen wären Kontrollregister in Perpheriebausteinen.

1.5.3 ZEITFUNKTIONEN

Der ESP8266 zählt wie jeder Computer die Zeit, die seit dem Einschalten vergangen ist mit Hilfe der Hardware. Die Funktion *millis()* liefert diese Zeit in Millisekunden (vgl. numerische Funktionen). Dieses BASIC enthält Zeitfunktionen, die auch Datum und Uhrzeit zur Verfügung stellen. Voraussetzung ist eine Verbindung mit einem AccessPoint (Router) und entsprechendem Internetzugang.

Systemzeit	*millis()*
Verzögerung	*delay*
Datum & Uhrzeit	*time()*
Unixzeit	*unixtime()*
Zeit Setup	*time.setup()*

Ohne eigenes Zutun holt sich BASIC die aktuelle Zeit aus dem Netz. Wie auch in anderen BASIC-Versionen liefert die Funktion *time()* das gewünschte Ergebnis.

print time()

Fri Dec 28 22:07:13 2018

Mit einem Format-String kann die Zeitausgabe angepasst werden:

print time(“year”)

print time(“month”)

print time(“day”)

print time(“dow day. month year”)

print time(“hour:min:sec”)

2018

Dec

28

Fri 28. Dec 2018

22:13:11

Done…

Um Monate nicht als amerikanischen Text, sondern als Zahl anzuzeigen, kann folgende Umwandlung Anwendung finden, die Zeichenketten-Funktionen benutzt. Die 2. Zeile mit der Datumszuweisung ist sehr lang und kann umgebrochen erscheinen. Das Listing besteht aus vier Zeilen (Zeilenumbruch in der Printfassung beachten).

m =
“Jan01Feb02Mar03Apr04May05Jun06Jul07Aug08Sep09Oct10Nov11Dec12”

datum = mid(time(),9,2) & “.” & mid(m, 3 + instr(m, mid(time(),5,3)),2) & “.” & mid(time(),21,4)

zeit = mid(time(),12,8)
print “am ” & datum & ” um ” & zeit

---

am 09.01.2019 um 09:25:40

Done…

Einige Systeme arbeiten mit der sogenannten Unix-Zeit. Sie beginnt am 1. Januar 1970, 0:00 UTC und gibt die seitdem vergangenen Sekunden an. Die Funktion *unixtime()* wandelt die Sekunden wieder in eine übliche Zeit. Formatierungen sind auch hier möglich.

print unixtime(0)

print unixtime(1559881200)

Thu Jan 01 00:00:00 1970

Fri Jun 07 04:20:16 2019

Done…

Die Zeit-Routinen, die auf Arduino-Bibliotheken beruhen, erlauben auch die Einstellung der Zeitzone und der Sommer- und Winterzeit. Der BASIC-Funktion *time.setup()* können bis zu drei Parameter übergeben werden. Der erste Parameter ist eine Zahl für die Zeitzone. Für Mitteleuropa ist das die ‚1'. Der zweite Parameter steht für die Sommer-/Winterzeit und der dritte Parameter ist ein optionaler Zeitserver. Im Winter muss für den Standort Berlin oder Hamburg *time.setup(1)* einmalig aufgerufen werden (MEZ: + 1 Stunde gegenüber UTC). Das System merkt sich diese Einstellung (Settings), so dass die Uhr zumindest im Winter richtig läuft, auch wenn nicht ständig diese Initialisierung erfolgt. Die beiden anderen Parameter werden in der Referenz nicht genau erläutert, lassen sich aber über den Quelltext des ESPBASIC und der benutzten Bibliothek untersuchen. Zumindest in der hier vorliegenden Version lief der voreingestellte Zeitserver problemlos, so dass diese Recherche bisher nicht notwendig erschien.

Ohne Internetzeit kann ein relativ genauer Sekundentakt mit der folgenden Routine erzeugt werden. Während des Programmlaufs reagiert das Browserfenster nicht. Die Intervalle sind deutlich genauer als mit einem *delay()* , wie die Ausgabe nach fünf Sekunden zeigt.

t0 = millis()

```
dt = 1000
t1 = t0
for i = 1 to 5
 print (millis()-t0)
 t1 = t1 + dt
 do
 loop while (millis()<t1)
next i
```

```
8
1000
2000
3000
4000
```

Done…

ESPBASIC verfügt über weitere spezielle Timer, die im zweiten Kapitel unter *Timer und Interrupts* zur Anwendung kommen. Ein Beispiel mit *delay* befindet sich bei den numerischen Funktionen bei *millis()*.

1.6 Verzweigungen und Unterprogramme

Bei programmtechnischen Abläufen kann es vorkommen, dass bei bestimmten Bedingungen der Programmlauf anders ablaufen soll. Drückt der Anwender eine Taste, so soll darauf reagiert werden. Dass Programm überprüft also in einer Schleife, ob eine Taste gedrückt wird. Wenn dies der Fall ist - also diese Bedingung erfüllt ist - verzweigt der Programmlauf. Auf unterster Maschinenebene sind dies Bitabfragen und anschließende Sprünge zu bestimmten Adressen, um dort das Maschinenprogramm weiter zu durchlaufen. BASIC kann das ganz ähnlich. In diesem Zusammenhang gibt es in den meisten Versionen und auch in ESP8266BASIC

Wenn … dann	*IF .. THEN*
Gehe zu	*GOTO*
Unterprogramm	*GOSUB/RETURN*

1.6.1 IF THEN ELSE ENDIF

Verzweigungen können erfolgen, wenn eine Bedingung eintritt. Die sonst gültige Bedingung kann mit *Else* berücksichtigt werden. Aus der Referenz des ESPBASIC-Autors, der eine Variable gerne *bla* nennt, ein erstes Beispiel. Es wird eine Variable auf den Wert 4 geprüft. Ist der Vergleich *bla = 4* Wahr, so verzweigt das Programm und liefert die Ausgabe „bla ist 4“, sonst wird „bla ist nicht 4“ ausgegeben.

```
If bla = 4 then
 print “bla ist 4”
Else
 print “bla ist nicht 4”
Endif
```

Auch Verschachtelungen sind möglich:

```
If a > b then
 print “a ist größer b”
 If b > c then
   print “b ist größer c”
 Else
  print “b ist nicht größer c”
 Endif
Else
 print “a ist nicht größer b”
Endif
```

Mit diesen Mitteln könnte nun das kleine Lottoprogramm von weiter oben überprüfen, ob eine Zahl schon gezogen wurde. Die Ziehung erfolgt mittels Abspeicherung in einem Array. Es wird solange „gezogen“ bis alle Zahlen unterschiedlich sind. Die Variable *f* erhöht sich, falls eine Zahl schon vorhanden ist.

```
Dim z(6)
n = 0
Do ‘Ziehung
 x = rnd(48)+1
 ‘Print x
 f = 0
 For i = 1 to n
   If z(i) = x  then f = f + 1
 Next i
 If f = 0 then
   n = n + 1
   z(n) = x
 Endif
Loop until n >= 6
Print “—- 6 aus 49 ——”
for i = 1 to 6
 Print z(i)
Next i
```

1.6.2 GOTO, GOSUB UND RETURN

Vermutlich weil BASIC eine der ältesten Programmiersprachen ist und aus der Zeit stammt, als man auch noch viel in reiner Maschinensprache programmierte gibt es den Programmsprung GOTO. Wurde in frühen Tagen dieser Sprache noch mit Zeilennummern gearbeitet und entsprechend zu einer Zeile gesprungen (GOTO 10), ist das bei Listings ohne Zeilennummern nur mit Sprungmarken bzw. Label möglich. ESPBASIC nennt eine Sprungmarke einen *Branch* . Dieser *Branch* in Klartext steht in eckigen Klammern. Programmabläufe mit *GOTO* -Anweisungen können extrem unübersichtlich werden und lassen sich fast ganz mit anderen Schleifenstrukturen und Verzweigungen umgehen. Zunächst soll der umgekehrte Weg beschritten werden, indem eine *FOR -NEXT* -Schleife nachgebildet wird, die von 1 bis 10 zählt. Das Beispiel zeigt gleichzeitig, dass der *GOTO* -Befehl auch im Abschnitt Wiederholstrukturen aufgeführt werden könnte.

```
i = 1
```

```
[nochmal]
print i
i = i + 1
if i <= 10 then goto [nochmal]
```

Formuliert man das Zeitbeispiel von weiter oben nicht mit DO LOOP UNTIL, sondern mit GOTO, hat das Listing den folgenden Aufbau. Das Ergebnis ist identisch.

```
t0 = millis()
dt = 1000
t1 = t0
for i = 1 to 5
 print (millis()-t0)
 t1 = t1 + dt

 [warten]
 if (millis()<t1) then GOTO [warten]
next i
```

Werden Programmteile öfter benötigt oder wiederholen sich Programmabschnitte, so können diese in Unterprogrammen ausgelagert werden. Ein Unterprogramm (Subroutine, Sub) führt Dinge aus und wird mit *GOSUB [branch]* aufgerufen. Am Ende sorgt ein *RETURN* dafür, dass der aufrufende Programmabschnitt an der Stelle fortgeführt wird, an der der Aufruf erfolgte. In Visual Basic nennen sich solche Unterprogramme Sub name() und enden mit einem End Sub. Indem man den Namen der Sub aufruft wird in VB das Unterprogramm ausgeführt. In ESPBASIC wird ein Unterprogramm mit einem *Branch* begonnen und kann darüber aufgerufen werden. Parameter können in ESPBASIC nicht, wie in VB, übergeben werden, es müssen globale Variablen Verwendung finden.

```
Print “Start Hauptprogramm”
Gosub [Unterprogramm]
Print “Noch Hauptprogramm”
Gosub [Unterprogramm]
Print “Ende Hauptprogramm”
end
[Unterprogramm]
```

Print “Text vom Unterprogramm”

Return

Start Hauptprogramm

Text vom Unterprogramm

Noch Hauptprogramm

Text vom Unterprogramm

Ende Hauptprogramm

Done…

Das Prinzip der Unterprogramme findet man auch auf der untersten Programmierebene, der Maschinensprache. Dort wird so ein Aufruf oft als ‚Call‘ bezeichnet, gefolgt von der Speicheradresse des Unterprogramms, was dort auch mit einer ‚Ret‘-Anweisung zum Aufrufer zurückkehrt.

Es folgen zwei Beispiele, die die Zweierpotenzen ausgeben. Zuerst die Lösung mit der mathematischen Potenz-Funktion, danach die maschinennahe Variante mit Bitschiebung.

```
bit = 1
for i = 1 to 8
 print bit
 gosub [zweihoch]
next i
end
[zweihoch]
bit = pow(2,i)
return
```

oder

```
bit = 1
for i = 1 to 8
 print bit
 gosub [zweihoch]
next i
end
```

```
[zweihoch]
bit = bit << 1
return
```

```
1
2
4
8
16
32
64
128
Done…
```

1.7 DATEIEN

Dateien speichern Daten und stellen diese zur späteren Verarbeitung bereit. ESPBASIC weicht hier etwas von anderen BASIC-Dialekten ab, dennoch lassen sich Daten in Dateien sichern. Das Dateisystem ist zum einen von der Benutzeroberfläche zugänglich, zum anderen auch über Programmierfunktionen. Die Daten liegen im selben Flash-Bereich wie die gespeicherten BASIC-Programme. Der programmtechnische Zugang zum Dateisystem erfolgt über fünf Funktionen oder Kommandos:

Lesen Zeichenkette	*Read()*
Lesen Zahl	*Read.Val()*
Schreiben Zeichenkette	*Write()*
Löschen	*Del.Dat()*

Programm laden	*Load*

Inhalte von Dateien sind Zeichenketten, also Text. Mit *Read.Val()* kann schon beim Einlesen eine Zeichenkette mit numerischen Zeichen in eine Zahl gewandelt werden. In einem Unterverzeichnis *„\data“* legt dieses BASIC einige Einstellungen – auch als Text – ab. Ein Unterverzeichnis *„\uploads“* wird angelegt, wenn über den BASIC-Filemanager Dateien zum ESP hoch geladen werden. Das können Bilder, Töne und auch JavaScript-Dateien sein, die z. B. weiter unten zur Anwendung kommen. BASIC-Programme als Textdateien liegen im Wurzelverzeichnis. Hier ein Blick auf das Dateisystem durch Aufruf des BASIC-Datei-Managers:

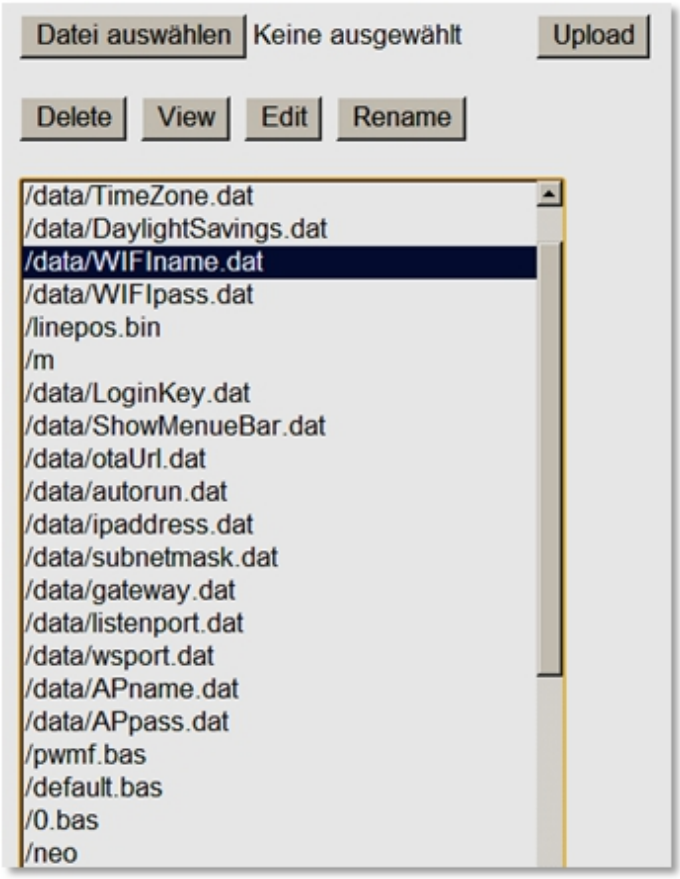


Abbildung 1-3: Dateien in ESPBASIC

Hier können manuell Dateien hochgeladen (Datei auswählen und Upload), gelöscht (Delete), umbenannt (Rename), geladen (Edit) und in den Editor geladen werden. Auch Zeiteinstellungen, wie Zeitzone und Router-Zugang, liegen hier unverschlüsselt (je nach Version) als Textdateien. Es sollte

darum in den Einstellungen besser ein Passwort vergeben werden, um das Dateisystem nicht jedem zugänglich zu machen. Auch versehentliches Löschen von Dateien und Programmen wird dadurch etwas erschwert. Dies ist besonders bei dem gleichzeitigen und gemeinsamen Zugriff von verschiedenen Browsern und Personen zu beachten. Dateien Löschen geht meist schneller als deren Erstellung und Humor hat viele Ebenen.

Als Test wird eine Datei mit dem Dateinamen „X1“ angelegt und der Zahlenwert 88 der Variablen *wx* abgelegt. Der Wert wird als Zeichenkette gespeichert. Danach erhält die Variable *rx* den Wert aus der Datei zurück. Zum Schluss wird die Datei gelöscht.

```
wx = 88
write(“x1”,x)
Print “Datei x1 mit ” & wx & ” gespeichert”
rx = read.val(“x1”)
Print “Wert “& rx & ” aus x1 gelesen”
Print “Datei x1 wird entfernt”
del.dat(“x1”)
```

Datei x1 mit 88 gespeichert

Wert 88 aus x1 gelesen

Datei x1 wird entfernt

Done…

Zur Speicherung mehrerer Zahlenwerte in einer Datei kann eine präparierte Zeichenkette zur Anwendung kommen. Die Zeichenkette *a$* erhält in einer Schleife die Werte 1 bis 5 zugeteilt, jeweils mit einem Leerzeichen getrennt. Nach der Speicherung dieser Zeichenkette erhält dann die zweite Variable *b$* den Dateiinhalt. Mit der Funktion *WORD* erhält man die durch Leerzeichen getrennten Einzelwerte wieder zurück und diese stehen dann zur Ausgabe bereit.

```
a$ = ””
for i = 1 to 5
 a$= a$ & str(i) & ” “
```

```
next i
‘ print a$
write(“x1”,a$)
b$ = read(“x1”)
for i = 1 to 5
 print word(b$,i)
next i
del.dat(“x1”)
```

1
2
3
4
5

Done…

Das Nachladen von Programmen erfolgt mit *Load* . Dabei ist die geänderte Syntax ohne Klammerung zu beachten. Soll das voreingestellte Programm von BASIC aus gestartet werden, so erreicht man dies z. B. durch die Zeile *load „\default.bas“* .

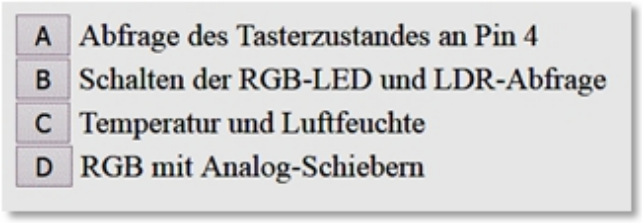


Abbildung 1-4: Programm Auswahl mit Load

Dadurch, dass Programme - oder Apps - mit *LOAD* nachgeladen werden können, besteht die Möglichkeit ein kleines Menü zu entwerfen. Dies erinnert etwas an die Anfänge der „Home-Computerei“, nach dem Motto “Nach Enter geht’s los”. Die Auswahl erfolgt heute jedoch in einem in C geschriebenem BASIC mittels eines JavaScript-Buttons in einem HTML5-Browser via WLAN. Bei entsprechenden Dateinamen kann ein Menü-Listing wie folgt aussehen:

```
cls
button “A”,[A]
wprint ” Abfrage des Tasterzustandes an Pin 4<br>”
button “B”,[B]
wprint ” Schalten der RGB-LED und LDR-Abfrage<br>”
button “C”,[C]
wprint ” Temperatur und Luftfeuchte<br>”
button “D”,[D]
wprint ” RGB mit Analog-Schiebern<br>”
wait
[A]
load “/din.bas”
[B]
load “/rgb.bas”
[C]
load “/DHT11.bas”
[D]
load “/rgbSlide.bas”
```

Die entsprechenden Programme müssen dazu lediglich über den Beenden-Knopf die Datei */menu.bas* laden, wenn obige Zeilen unter diesem Namen abgespeichert wurden. Auch ein gelöschter Bildschirm via *cls* als erste Anweisung macht die Dinge übersichtlich.

```
[ende]
wprint “Menue wird aufgerufen.”
delay 3000
load “/menu.bas”
```

Die Button-Elemente als Teil des sogenannten Web-Interfaces von ESPBASIC sind unter anderem Gegenstand des folgenden Abschnitts.

---

2 ESPBASIC SPEZIELL

ESPBASIC wurde speziell für den ESP8266 entwickelt und berücksichtigt dessen Eigenschaften in besonderer Weise. Wurden in Kapitel 1 allgemeine BASIC Eigenschaften mit kurzen Beispielen aufgeführt, sollen in diesem Abschnitt die nur bei dieser Hardware interessanten Aufrufe vorgestellt werden. Ursprünglich wurde der ESP8266-Baustein als Seriell-WiFi-Konverter bezeichnet, also ein Chip, um serielle Daten über WLAN zu senden und umgekehrt. Externe serielle Daten gelangen über die Anschlüsse RX/TX zum ESP. Aber auch anderen Ein- und Ausgänge vom ESP und deren Anwendung werden in vielfacher Weise unterstützt. Die sich durch WLAN-Anbindung ergebenden Möglichkeiten lassen sich in diese Bereiche einteilen:

- Serielle Daten Rx/Tx, Serieller Port 2
- Hardware Interface (Digital/Analog)
- Timer und Interrups
- WiFi Funktionen
- TCP/IP und HTTP-Client – Telnet, Browser, Mail
- UDP/IP Server
- Web Interface, Web Graphics

2.1 Serielle Kommunikation

Jeder ESP8266 verfügt über zwei serielle Leitungen RX/TX (Empfangen/Senden). Als der ESP-01 erschien, war das der einzige Weg mit dem Chip in Kontakt zu treten. Diese Leitungen führen 3,3 Volt Pegel, entsprechen also nicht dem TTL-Pegel von 5 Volt und schon gar nicht dem +/- 12 Volt Pegel einer RS232-Schnittstelle. Für den Anschluss solcher Hardware müssen Anpassungen erfolgen. Im Kapitel Anwendungen sind Beispiele dazu zu finden. An dieser Stelle soll nur die grundsätzliche Funktionalität der Routinen gezeigt werden.

2.1.1 Hardware Serial

Serialprint

SerialprintLn

Baudrate

Input

Serialinput

SerialFlush

SerialTimeout

Serial.read.int()

Serial.read.chr()

Serial.available()

Als Hardware wird ein ESP8266 Witty Cloud oder ein D1 Mini Board mit integriertem USB-Konverter benutzt, worüber auch die Spannungsversorgung erfolgt. Der Konverter wandelt die seriellen Daten so um, dass ein angeschlossener Rechner oder ein Smartphone mit OTG-Kabel direkt mit dem ESP kommunizieren kann. Die *Print* -Anweisung liefert die Ausgaben an beide Ausgabekanäle, den seriellen Ausgang und das Browser-Fenster. Soll die Kommunikation mit einem Windows-PC oder Tablet erfolgen, so ist der einfachste und schnellste Weg die Oberfläche des Installationsprogramms zu benutzen, welches im Anhang dargestellt wird und für die Erstinstallation von ESPBASIC benötigt wird. Falls ein Smartphone benutzt werden soll, ist eine entsprechende App aus dem Store zu installieren. Der folgende Abschnitt benutzt ein Android-Phone mit OTG-Kabel und die App „Serial USB Terminal 1.21“ aus dem Playstore.

Nun funktioniert auch der einfache INPUT-Befehl. Es entsteht ein typischer BASIC-Dialog mit einem Prompt „Bitte um

Eingabe>“. Der Benutzer kann nun eine Eingabe tätigen und mit „Enter“ abschicken. Im Terminalprogramm muss eingestellt sein, dass die Eingabetaste auch ein CR/LF sendet. Das kann meist in den Einstellungen geändert werden. In dieser App sieht das so aus:

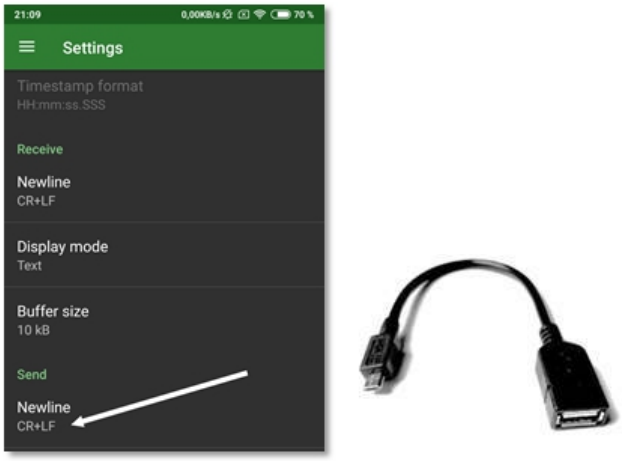


Abbildung 2-1: Einstellungen für das serielle Terminal und OTG-Kabel

Das Listing im Browser besteht aus zwei Zeilen:

input “Bitte um Eingabe”, a

print “Empfangen: ” & a

Beim Start des Programms sieht man im Browser nichts. In der App erscheint der Prompt. Wenn z. B. die Zahl 898 eingegeben und gesendet wird, sehen das Browserfenster und das Terminalfenster wie folgt aus:

Empfangen: 898

Done…

Wie man erkennt, werden auch die Speichervorgänge über die serielle Schnittstelle ausgegeben. In diesem Terminalfenster stehen noch mehr Informationen. Ganz oben teilt der ESP selber mit was er gerade macht, allerdings in einer anderen Geschwindigkeit und darum nicht lesbar.

Abbildung 2-2: Informationen über die serielle Schnittstelle

Stellt man die Baudrate im Terminal auf 76800 hat man gute Aussicht auf Erfolg diese Meldung im Klartext zu lesen. Dann meldet ESPBASIC, dass der Server auf Port 80 lauscht, eine ältere Version gestartet wurde und dann noch die MAC vom diesem ESP8266. Es folgt der Verbindungsversuch mit dem eingestellten Router, der nach 3 Zählern (1,5 Sekunden) gelingt. Die IP, die dem ESP zugeteilt wurde, wird dargestellt. Die muss in die Adresszeile des Browsers eingegeben werden, um die BASIC-Umgebung zu erhalten. Name und Password des Routers wird unterdrückt und ESPBASIC meldet „Done…“. Der Autor hat das Testprogramm unter „/hallo“ erfolgreich abgespeichert. Danach wurde mit RUN das Programm gestartet. Da ein lokales Echo ausgeschaltet ist, zeigt das Terminal die Eingabe noch nicht, aber im Browser sieht man, dass die drei Ziffern angekommen sind.

Sollen die Ausgaben nur am seriellen Anschluss erscheinen, so erfolgt dies mit den Kommandos *SerialPrint* und *SerialPrintLN* , einmal ohne -, und einmal mit CR/LF (Zeilenvorschub). Ein Beispiel:

wprint “dieser Text geht zum Browser”

serialprintln “dieser Text geht nur zum Terminal”

Die Routine *Serialinput* liefert die Empfangsdaten der RX-Leitung des ESP8266. Der Aufruf

serialinput a

print a

endet sofort mit einem „Done.“, da nicht sehr lange – wie bei *INPUT* - auf eine Eingabe gewartet wird. Abhilfe könnte die Anweisung *SerialTimeout* mit einem Parameter in Millisekunden schaffen. Der Test mit *SerialTimeout* 5000 gelingt nicht. Nach genauerem Lesen der Referenz bezieht sich diese Anweisung auf den *INPUT* -Befehl, der dann nur 5 Sekunden warten würde. *SerialFlush* löscht alle Zeichen im Empfangspuffer, so dass irgendwelche nicht abgeholte Zeichen gelöscht werden.

ESPBASIC benutzt eine sehr elegante Art und Weise mit der Empfangsproblematik umzugehen. Da ein BASIC-Programm nicht ständig abfragen möchte, ob Zeichen im Empfangspuffer neu eingegangen sind, gibt es einen *Branch* , der angesprungen wird, wenn das System ein Zeichen empfängt. Dieser *Branch* (Sprungziel) wird mit *SerialBranch* festgelegt. An dieser Sprungmarke kann dann ein Zeichen in Empfang genommen werden. Ein typischer Fall, wenn zeitlich zufällig Zeichen an der Leitung RX erscheinen:

```
baudrate 9600
serialbranch [rx]
wait
[rx]
serialinput a
print “RX: ” & a
return
```

RX: moin

RX: 0815

Nach dem Programmstart sieht man nichts, der Browser ist jedoch ansprechbar. Werden im Terminal Zeichen abgeschickt, so wird der Programmteil bei Sprungmarke [*rx*] ausgeführt. Da Zeichen im Puffer gemeldet wurden, können sie mit *Serialinput* gelesen werden. Zur Kontrolle erfolgt die Ausgabe auf beiden Kanälen. Ein *RETURN* schließt die Aktion ab. ESPBASIC wartet weiter auf Empfangsdaten, kann aber in der Zwischenzeit andere Dinge erledigen. Hier folgt ein Beispiel mit dem *Timer* , um die Ereignissteuerung in ESPBASIC weiter zu verdeutlichen.

Während auf eingehende Zeichen gewartet wird, soll jede Sekunde eine Zahl um eins erhöht und auf beiden Kanälen ausgegeben werden. Der *Timer* wird ähnlich wie der *SerialBranch* eingerichtet. Für ein Intervall von 1000 ms initiiert die Zeile *timer 1000, [tm]* , dass die Routine bei *[tm]* einmal pro Sekunde ausgeführt wird. Man kann erkennen, dass im Terminal die Zahlen weiter erhöht werden, auch wenn im Browser mit dem Zurück-Button wieder im Editor weiter programmiert wird. Erst ein erneutes Speichern oder ein Reset beenden den Programmlauf.

```
timer 1000, [tm]
serialbranch [rx]
wait
[rx]
serialinput a
print “RX: ” & a
return
[tm]
print i
i = i + 1
wait
```

Mit diesen Zeilen lassen sich auch die weiteren seriellen Kommandos testen. Die Funktion *Serial.available()* liefert die Anzahl der empfangenen Zeichen (im Puffer) und mit *Serial.read.int()* erhält man das Byte als Zahl, mit *Serial.read.chr()* als Zeichen. Im folgenden Beispiel läuft der *Timer* wie gehabt, während im Terminal „0815“ abgeschickt wird. Die Empfangsroutine wird angesprungen, wenn Empfang vorliegt. Nun werden solange Bytes aus dem Empfangspuffer als Zahlenwert gelesen und ausgegeben, wie welche vorhanden (available) sind. Das Ergebnis ist, dass die fortlaufenden Zahlen des Timers unterbrochen werden mit der Zahlenfolge 48, 56, 49, 53, 13, 10. Die ersten vier Zahlen sind der ASCII-Code der eingegebenen Zeichenfolge, danach eine 13 für CR (Wagenrücklauf/CarriageReturn) und 10 für LF (Zeilenvorschub/LineFeed).

```
timer 1000, [tm]
serialbranch [rx]
```

```
wait
[rx]
do
 print serial.read.chr()
loop while serial.available > 0
return
[tm]
print i
i = i + 1
wait
```

Laut Forum soll der Befehl *baudspeed* erweiterte Baudrateneinstellungen erlauben. Möglicherweise ist auch der Originalbefehl *baudrate* gemeint. Als Beispiel wird *baudspeed 9600 5N1* angegeben mit den weiteren Möglichkeiten 5N1 6N1 7N1 8N1 5N2 6N2 6N2 7N2 8N2 5E1 6E1 7E1 8E1 5E2 6E2 7E2 8E2 5O1 6O1 7O1 8O1 5O2 6O2 7O2 8O2. Entsprechende Tests stehen noch aus.

2.1.2 SOFTWARE SERIAL

Da die Hauptkommunikation über die festen Leitungen RX/TX des Moduls erfolgt, kann es nützlich sein einen zusätzlichen Kommunikationskanal zu betreiben. Die folgenden Routinen können über fast beliebige Anschlüsse (Pins) serielle Daten senden und empfangen.

Serial2begin

Serial2end

Serial2print

Serial2println

Serial2input

Serial2Branch

Mit *Serial2begin* wird das zweite System initialisiert. Parameter sind Baudrate, TX-Pin, RX-Pin. Soll mit 9600 Baud über die Pins 5 und 4 kommuniziert werden, um z. B. mit einem HC06-Bluetooth-Modul zu sprechen, so lautet die Initialisierung *Serial2begin 9600, 5, 4* . *Serial2end* gibt die benutzen Ressourcen wieder frei. *SerialPrint* , *Serial2PrintLn* und *Serial2Input* verhalten sich überwiegend ähnlich wie die Hardwarevarianten. Auch ein *Serial2Branch* ist vorgesehen, wodurch eigenes Polling entfällt und ereignisorientiert eintreffende Daten abgeholt werden können.

2.2 TIMER UND INTERRUPS

Unterbrechungen des laufenden Programms bei bestimmten Ereignissen werden Interrupts genannt. Auch regelmäßig auftretende Aktionen müssen den Programmlauf unterbrechen und können durch sogenannte Timer gesteuert werden. Der ESP8266 wird an dieser Stelle durch BASIC etwa sieben Aufrufe unterstützt. Einige benutzen dabei Sprungmarken, die ESPBASIC *Branches* nennt.

SerialBranch
Serial2Branch
Timer
TimerCB
Reboot
Sleep
Interrupt

Die beiden ersten Anweisungen gehören zur seriellen Kommunikation und sind im vorigen Abschnitt erläutert worden. Der *Timer* kam auch schon zum Einsatz, soll aber hier nochmals überprüft werden.

2.2.1 TIMER

Soll in regelmäßigen Intervallen eine Aktion wie z.B. eine Messung erfolgen, ist ein *Timer* die erste Wahl. Es sollen Millisekunden zur Anzeige kommen. Dabei wird die Systemzeit über *millis()* benutzt, um die Abweichungen im Millisekunden-Bereich zu erfassen. Die Systemzeit beginnt bei null nach dem Boot-Vorgang und zählt die Laufzeit des ESP in Millisekunden. An anderer Stelle kamen andere Zeit-Routinen zum Einsatz. Hier folgt die *Timer* -Variante. Da die Interpretation der Initialisierung etwas Zeit benötigt, wird *t0* beim ersten *Timer* -Aufruf mit der Startzeit versehen. Die Ausgabe zeigt recht gute Ergebnisse:

```
t0 = 0
timer 1000,[tm]
wait
[tm]
t = millis()
if t0 = 0 then t0 = t
print t-t0
wait
```

Die Vergleichsvariable bekommt den Wert 0, um in der *Timer* -Routine beim ersten Aufruf die aktuelle Systemzeit darin zu speichern, die dann bei folgenden Aufrufen jeweils von der fortgeschrittenen Zeit subtrahiert wird. Die Ausgabe zeigt die Sekunden in ms-Auflösung.

1000
2000
3000
4000
5001
6000
7000

8000

9000

10001

Ein 0-Intervall beim *Timer* -Aufruf stoppt den *Timer* . Auf diese Art und Weise erfolgt im nächsten Listing nur die Anzeige der ersten 10 Sekunden.

```
t0 = 0
timer 1000,[tm]
wait
[tm]
t1 = millis()
if t0 = 0 then t0 = t1
t = t1 - t0
print t
if t > 9000 then timer 0
wait
```

Der zweite *Timer* nennt sich *TimerCB* und steht für ‚CallBranch'. Damit ist es möglich Unterprogramme per Timer auf zu rufen. Das spart den Befehl *Gosub* , dafür muss aber der Kode am *TimerCB* -Branch mit einem *Return* enden. Zwei unabhängige Timer sind sehr gute Programmierwerkzeuge in diesem Kontext, erlauben sie doch zeitlich getrenntes Messen und Steuern. Hier ein kurzer Test:

```
t0 = 0
timer 500,[tm]
timercb 1000,[unterprogramm]
wait
[tm]
t1 = millis()
if t0 = 0 then t0 = t1
t = t1 - t0
print t
if t > 4000 then
  timer 0
```

timercb 0

endif

wait

[unterprogramm]

print time(“hour:min:sec”)

return

500

13:29:28

1000

1500

13:29:29

2000

2500

13:29:30

3000

3500

13:29:31

4000

4500

2.2.2 REBOOT

Ein Reset oder Neustart kann im Browser über die Einstellungen (Settings) unten auf der Seite erfolgen, oder eben auch von ESPBASIC aus mit der *Reboot* -Anweisung. Es gibt keine Parameter; das System startet neu. Der Browser verliert die Verbindung bis die WiFi-Verbindung wieder steht. Ist ein *„/default.bas* “ Programm aktiviert, so startet dies nach 30 Sekunden.

2.2.3 *SLEEP*

Mit *Sleep* wird der Schlaf-Modus eingeschaltet, wodurch die Stromaufnahme reduziert werden kann. Je nach Board bewegt sich die Stromaufnahme bei inaktiven Wifi und ohne Spannungsregler und LED im Mikroampere-Bereich. Dem Befehl wird als Parameter eine Zeit in Sekunden übergeben, sie legt fest wie lange geschlummert werden soll. Der ESP führt nach dieser Zeit ein Reboot aus, so dass nach 30 Sekunden wieder *„/default.bas“* anläuft. Zur Datensicherung in den Wach-Phasen bieten sich Dateien oder kurze Wifi-Aktivitäten an. Ein Test-Aufruf gestaltet sich einfach mit der Zeile Sleep 60 für ein Nickerchen von einer Minute. Leider scheitert der Versuch mit ESPBASIC 3.0 auf dem Witty Cloud mit einem Crash und anschließendem Reboot. Ein erneuter Versuch mit einem „frischen“ ESP8266 wäre es wert die Dinge nochmals zu überprüfen.

2.2.4 *INTERRUPT*

Ein Interrupt in ESPBASIC wird ausgelöst, wenn sich an einem angegebenen Pin etwas ändert. Da am Witty Cloud der Taster mit Pin 4 verbunden ist, soll ein Test damit erfolgen. Der Programmierer muss sich also nicht selber darum kümmern, sondern wird quasi benachrichtigt, wenn an dem Anschluss Änderungen auftreten. Wie üblich ist ein *Branch* erforderlich und eine Pin-Angabe. Für die Überwachung eines an Pin 4 angeschlossenen Tasters, der zwischen Vcc und Gnd wechselt, funktioniert der folgende Ablauf, der beim Drücken und Loslassen das Wort „Wechsel“ ausgibt:

```
interrupt 4,[wechsel]
wait
[wechsel]
print “Wechsel”
wait
```

An Pin 15 ist die rote LED angeschlossen. Als kleine Spielerei soll diese LED bei jedem Tastendruck den Zustand wechseln. Da nur bei jedem zweiten Wechsel die LED wechseln soll, gibt es zwei Variablen mit zwei Negationen:

```
a = 0
```

```
b = 0
interrupt 4,[wechsel]
wait
[wechsel]
if b then
 io(po,15,a)
 a = not a
endif
b = not b
wait
```

2.3 HARDWARE I/O INTERFACE

Dieser Abschnitt befasst sich mit Ein- und Ausgaben über die Pins des ESP8266. Die GPIO (General Purpose Input Output) sind auf den verschiedenen Boards unterschiedlich nummeriert oder bezeichnet. Ein „PinOut“ oder Pinlayout des eigenen ESP sollte bereit liegen. Das hier verwendete Board Witty Cloud hat die Anschlüsse RXD, TXD, VCC (5V!), GND, REST, ADC, CH_PO und die GPIO00(D3), 02(D4), 04(D2), 05(D1), 13(D7), 14(D5), 15(D8), 16(D0). Die Anschlüsse Reset und CH_PO dienen der Programmübertragung und werden u.a. vom Installationsprogramm benutzt, um das ESPBASIC zu „flashen“. ADC ist der einzige Analogeingang, der mit Vorwiderstand und LDR helligkeitsproportionale Analogwerte liefert. An den Pins 12, 13, 15 sind die drei farbigen LED angeschlossen. Die kleine, blaue LED liegt an Pin 2. An 0 und 2 werden oft I2C Geräte betrieben. Pin 4(D2) und 5(D1) bieten sich für einen *SoftwareSerial* Anschluss an. Außer einiger besonderer Anweisungen beginnen die Kommandos mit *io* für Input/Output:

Portausgabe	*io po*
Porteingabe	*io pi*
PWM Ausgabe	*io pwo*

PWM Eingabe	*io pwi*
Servomotor	*io servo*
Analogeingabe	*io ai*
Zustandabfrage	*io laststat*
PWM Frequenz	*pwmfreq*
Temperatursensor	*temp*
Pin1 Reset	*gpio1reset*

2.3.1 DIGITAL AUSGABE

Wo Leuchtdioden sind, ist ein Blink nicht weit. Darum zuerst ein Sekundenblink in ESPBASIC. Es wird der *Timer* benutzt, der weiter oben bereits angewandt wurde. Das vorige Taster-Beispiel bedarf nur einer geringfügigen Anpassung, es wird die kleine, blaue LED an Pin 2 gesteuert.

```
a = 0
timer 500,[wechsel]
wait
[wechsel]
io(po,2,a)
a = not a
wait
```

Das Blinkprogramm könnte auch mit *delay* und einer Endlosschleife programmiert werden, aber dann wäre der Browser abgehängt. Wer das nicht glaubt, der probiere das folgende Listing im Arduino-Stil mit einer Endlosschleife.

```
LEDpin = 2
Do
  io(po,LEDpin,1)
  delay 500
  io(po,LEDpin,0)
  delay 500
Loop until 0
```

2.3.2 DIGITAL EINGABE

Mit PortIn (*pi*) lässt sich der Zustand eines Digitaleingangs abfragen. Weiter oben steht ein Beispiel zur Taster Abfrage über Interrupt-Behandlung. Es geht auch mit einem direkten Kommando. Das Beispiel klemmt den Browser ebenfalls ab, da keine Ereignisabfrage eingesetzt wird. Um dennoch die Funktion zu prüfen, wird die LED an Pin 2 benutzt, die bei gedrücktem Taster an Pin 4 leuchten soll. Da der Taster den Eingang auf Masse zieht, ist das Ergebnis invers:

```
LEDpin = 2
Do
```

```
  taster = io(pi,4)

   io(po,LEDpin,taster)

  delay 20
Loop until 0
```

Es geht noch kürzer:

```
Do
 io(po,2,io(pi,4))
Loop until 0
```

2.3.3 PWM AUSGABE

Die PWM-Pulsbreiten-Modulation benutzt eine Rechteckfrequenz mit unterschiedlichen Verhältnissen der Ein- und Ausschaltdauer. Ist das Rechtecksignal am Pin halb so lange an, wie aus und wiederholt sich dieser Vorgang schnell, entspricht der Spannungswert quasi einem Analogsignal der halben Spannungshöhe. Die Frequenz dieses Rechtecks ist fest vorgegeben, kann aber in Grenzen verändert werden. Eine Analogausgabe im Bereich 0 bis 1023 ändert also nur dieses Verhältnis und wäre bei 511 auf halbem arithmetischem Mittelwert der Rechteckamplitude. Eine LED leuchtet entsprechend unterschiedlich hell. Als Test sollen zwei unterschiedliche Ansätze zur Anwendung kommen. Einmal die Browser-Abklemm- oder Arduino-Stil-Variante und die ereignisgesteuerte Timer-Variante à la ESPBASIC. Die rote LED an Pin 15 des Witty Cloud soll langsam an und ausgehen, also pulsieren. Dazu wird der Wert 0 bis 1023 in einer Schleife ausgegeben, danach wird in einer Schleife zurück gezählt und ausgegeben. Die gesamte Aktion soll in der ersten Variante nur zehn Mal stattfinden, damit danach der Browser wieder reagiert:

Variante 1

```
LED = 15
for n = 1 to 10
 for i = 1 to 1023
  io(pwo,LED,i)
 next i
 for i = 1022 to 0 step -1
```

```
  io(pwo,LED,i)
 next i
next n
```

Variante 2

```
LED = 15
dx = 1
x = 0
timer 50,[tm]
wait
[tm]
io(pwo,LED,x)
x = x + dx
if (x < 0) or (x > 1023) then dx = -dx
wait
```

2.3.4 PWM-FREQUENZ

Ein Oszilloskop könnte die Signale darstellen und die verschiedenen Verhältnisse zeigen. Mit Hilfe eines Piezo-Beepers, der am LED Pin angeschlossen ist, hört man ein Ton konstanter Frequenz, die Änderung des Rechteckverhältnisses kann man dennoch wahrnehmen. Die Frequenzvariation umfasst den Bereich 1 bis 8000 Hertz und wird mit der Anweisung *pwmfreq* eingestellt. Eine zusätzliche Zeile zwischen *[tm]* und *io(pwo,LED,x)* kann die Tonausgabe deutlich beeinflussen, sie lautet:

```
pwmfreq 400+10*x
```

Damit ändert sich die Rechteckfrequenz sowohl im An-/Aus-Verhältnis als auch in der Gesamtdauer eines Zyklus. Der Bereich berechnet sich hier zu 400 Hz bis 2950 Hz. Es ist somit recht einfach mit einem solchen Beeper ein BASIC-Beep zu erzeugen. Zu beachten ist auch, dass die Frequenz erhalten bleibt, auch wenn das Programm weiter läuft. Das wird deutlich, wenn im nachfolgenden Listing das Intervall probehalber 1000 ms beträgt. Hier wird nun die Basis für ESP-Musik gelegt. Die Tonleiter ertönt am Beeper mit eingestellten 200 ms-Intervallen. Da der Beeper auch in seiner Lautstärke durch das Verhältnis beeinflussbar ist, steht der ersten Fuge,

auch mit entsprechend sensiblem Anschlag, nichts mehr im Weg.

```
LED = 15
A$=“523 587 659 698 784 880 988 1047”
x = 0
timer 200,[tm]
wait
[tm]
pwmfreq val(word(A$,x))
io(pwo,LED,5)
x = x + 1
if (x > 8) then x = 1
wait
```

Wem ein Kinderlied auch genügt, der probiere diesen Sound. Hinweis: ESPBASIC unterscheidet bei Variablen zwischen Groß- und Kleinschreibung!

```
LED = 15
Freq$ = “523 587 659 698 784 880 988 1047 08”
Note$ = “cdefgahCp”
Lied$ = “cdefggggaaaaggggaaaaggggpp”
Lx = 1
timer 500,[tm]
wait
[tm]
Ix = instr(Note$,mid(Lied$,Lx,1))
pwmfreq val(word(Freq$,ix))
io(pwo,LED,127)
Lx = Lx + 1
if (Lx > len(Lied$)) then Lx = 1
wait
```

2.3.5 PWM EINGABE

Der umgekehrte Weg soll mit *io(pwi,pin)* funktionieren. Das erwartete Ergebnis blieb hier bisher aus, möglicherweise wurde die Referenz falsch gedeutet. Eventuell gibt der Quelltext von ESPBASIC selber weitere Auskunft.

2.3.6 SERVOMOTOR

Ein Servomotor lässt sich mit ESPBASIC sehr leicht steuern. Mit der Anweisung *po(servo, pin, winkel)* stellt sich ein Servomotor an dem Pin auf den Winkel ein, der sich im Bereich von 0 bis 180 Grad befinden muss. Ein kurzer Test realisiert eine Art Scheibenwischer. Der Servo an Pin 15 wird auf 180 Grad gestellt und verharrt dort zwei Sekunden. Dann dreht er zurück auf null und wartet erneut, bis der Zyklus von vorn beginnt.

```
[Start]
Io(servo,15,180)
Delay 2000
Io(servo,15,0)
Delay 2000
Goto [Start]
```

Im Zusammenhang mit dem nun folgenden Analogeingang und dem lichtempfindlichen Widerstand des Witty Cloud könnte ein „Lampenoszillator“ konstruiert werden. Immer wenn der LDR hell beschienen wird, dreht der Servomotor so, dass ein Schalter oder sogar ein Dimmer die Lampe abdunkelt und dadurch die LDR Abfrage den Servomotor wieder in die andere Richtung schickt. Wieder etwas, was die Welt nicht braucht, aber Spaß macht.

2.3.7 ANALOGER EINGANG

Ein ESP8266-12 verfügt über einen ADC-Anschluss. Es ist ein Analog-Digital-Wandler, der in 1025(!) Stufen auflöst. Bei einer Bord Spannung von 3,3 Volt entspricht diese 10-Bit-Auflösung einer Spannungsauflösung von 3,2 mV pro Bit. Das Witty Cloud-Modul hat dort einen Spannungsteiler aus Festwiderstand und LDR verschaltet, so dass ein

helligkeitsproportionaler Analogwert messbar ist. Bei Tageslicht sind das meist Werte um 1025, während am Winterabend bei Innenraumbeleuchtung folgende Werte gemessen werden können:

616
602
601
432
380
629
617
600
577
613

Done…

Die Werte resultieren aus diesen Zeilen und etwas Handwedeln:

```
for i = 1 to 10
 print io(ai)
 delay 1000
next i
```

Soll die Spannung am Eingang im Browser oder am seriellen Ausgang erscheinen, so ändert sich das Listing zu:

```
for i = 1 to 10
 print io(ai,A0)/1024*3.3
 delay 1000
next i
```

2.3.8 STATUSABFRAGE – LASTSTAT

Ohne den Zustand eines Digitalausgangs zu ändern, kann mit *io(laststat, pin)* der Zustand des angegebenen Pins bzw. Anschlusses erfragt werden. Bei Steuerungen über das Internet kann man so erfahren, ob das Kommando der Ausgabe inzwischen angekommen ist. Bei langsamen Verbindungen kann zwischen dem Ausgabekommando und dem tatsächlichen Wechsel am Ausgang einige Zeit vergehen. Auch bei einem Touch-Interface ist eine Rückmeldung mit dem tatsächlichen Zustand durchaus hilfreich.

Die in der ESPBASIC-Referenz aufgeführten Funktion *Gpio1Reset* initialisiert TX Pin 1 erneut. Die Funktion *temp* unterstützt einen Temperatursensor durch eine eingebundene Hardware-Bibliothek und ist in Abschnitt 3 als Dallas-Sensor aufgeführt.

2.4 WiFi Funktionen

Der ESP8266 Baustein ist für Wifi-Betrieb entwickelt worden. Als die Heimautomatisierung per Wifi begann, veränderte dieser Baustein die Szene völlig. Inzwischen ist dieser Baustein mit seinen Möglichkeiten auch in Lichtschaltern und Steckdosen zu finden. Mit ESPBASIC kann sein Wirken auf sehr bequeme und einfache Art vom Browser aus mit wenigen Programmzeilen gestaltet werden.

Verbinden	*Wifi.Connect()*
Access Point	*Wifi.AP()*
Ausschalten	*WifiOFF*
Access Point und Station	*WifiAPSTA*
Router Scannen	*WifiScan()*
Router Kennung	*WifiSSID()*
Signalstärke	*WifiRSSI()*

MAC-Adresse	*WifiBSSID()*
Lokale IPv4	*ip()*

Wenn ein ESP8266 mit Spannung versorgt wird, startet ESPBASIC und versucht sich mit einen Wifi-Netzwerk zu verbinden. Wird kein bekanntes Netzwerk gefunden, oder die Verbindung scheitert, so spannt der ESP ein eigenes WLAN als Access Point (AP) auf. Ein Anwender verbindet sich dann mit diesem Netzwerk („ESPxxxx“) und gibt im Browser die voreingestellte IP 192.168.4.1 ein. Damit ist ESPBASIC immer erreichbar.

In den Einstellungen können diese Vorgaben geändert werden:

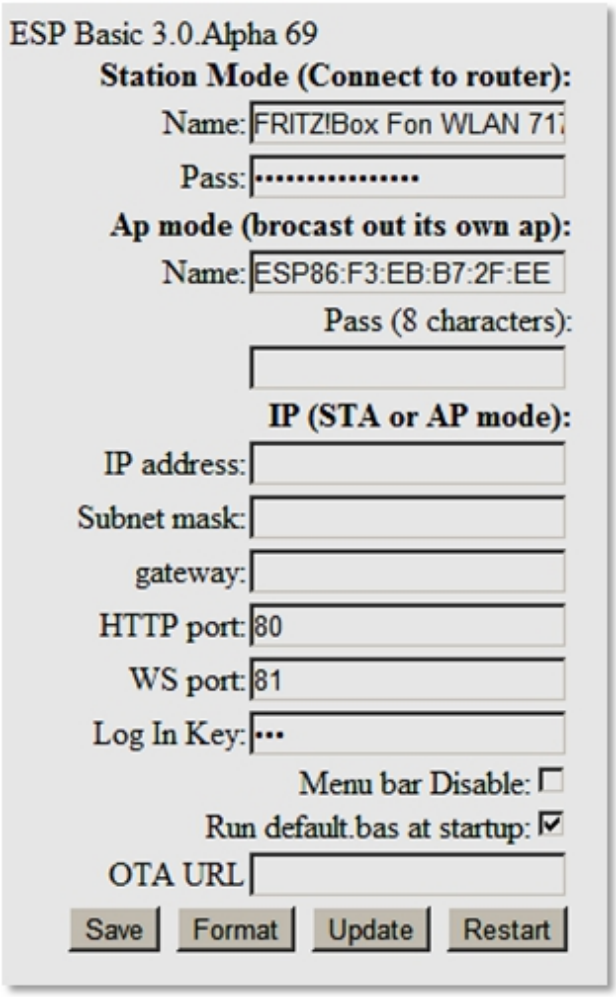


Abbildung 2-3: Einstellungen unter Settings

Der Befehlsvorrat von ESPBASIC bietet auch Möglichkeiten diese Verbindungen zu untersuchen oder auch zu ändern.

2.4.1 WiFiScan, SSID, RSSI, BSSID

Um mit dem ESP8266 die umgebenden Drahtlosnetzwerke zu erkunden, ist die Funktion *Wifi.Scan()* vorgesehen. Der Rückgabewert ist die Anzahl der gefundenen Netzwerke:

```
print wifi.scan() & ” Netzwerke empfangen”
```

---

4 Netzwerke empfangen

Done…

Die Funktion macht aber noch viel mehr. Sie legt die gefundenen Netzwerkinformationen in einem Array ab, die über einen Index (Zahl) angesprochen werden können. Auf diese Art und Weise erhält man die Kennung mit der Funktion *Wifi.SSID(Index)*. Ist ein Netzwerk vorhanden, so erfolgt die Abfrage des ersten Netzwerks mit *Wifi.SSID(1)* . Die Signalstärke erfährt man durch den Aufruf *WiFi.RSSI(1)*. Sollen alle Netzwerke mit Kennung und Signalstärke zur Anzeige kommen, so erfüllen die folgenden Zeilen diesen Zweck:

```
n = wifi.scan()
print n & ” Netzwerke im Bereich:”
for i = 1 to n
 print i & “. ” & wifi.ssid(i) & ” mit ” & wifi.rssi(i) & ” dB”
next i
end
```

---

5 Netzwerke im Bereich:

---

1. FRITZ!Box Fon WLAN 7390 mit -76 dB

---

2. UPC248F2AD mit -87 dB

---

3. FRITZ!Box Fon WLAN 7170 mit -54 dB

---

4. FRITZ!Box Fon WLAN 7240 mit -91 dB

---

5. EasyBox-BA8638 mit -92 dB

Done…

Zusätzlich liefert *WifiBSSID()* noch die MAC-Adresse z.B. des Routers.

2.4.2 WiFi Off

Soll WiFi ausgeschaltet werden, um möglicherweise Energie einzusparen, kann das einfach mit der Anweisung *wifioff* erfolgen. Strommessungen am Witty Cloud mit seinem vielen Ballast änderte an den 60 mA einer 5 Volt Powerbank nicht viel. Um nicht manuell zur Reset-Taste greifen zu müssen, soll ein Neustart nach 20 Sekunden erfolgen.

wifioff

delay 20000

reboot

oder

wifioff

timer 20000,[tm]

wait

[tm]

reboot

wait

in beiden Fällen mischt sich Google ein mit „*Ups! Google Chrome konnte keine Verbindung zu 192.168.178.39 herstellen.* “

2.4.3 WiFi AP

Ein Access Point (AP) ist ein Hot Spot oder ein Router, manchmal auch ein Smartphone mit eigenem Hot Spot. Wird ein ESP daheim im lokalen Netzwerk eingebunden, ist ein eigener AP nur erforderlich, wenn die Verbindung zum Router aus irgendwelchen Gründen nicht klappt. Wenn doch aus dem Programm heraus ein eigener AP eingeschaltet werden muss, steht der Befehl *wifiap()* zur Verfügung. Soll der ESP8266 als AP mit der Kennung „Keine Fritz!Box“ in den Äther gehen und kein Passwort nötig sein, wie zum Beispiel in der Wüste oder in Bielefeld, so ist der Aufruf: *wifiap(„Keine Fritz!Box“)* , oder mit Passwort *wifiap(„Keine Fritz!Box“, “Fritz“)* . Auf

einem Netbook unter Windows könnte das aussehen, wie hier darestellt.

Abbildung 2-4: PC findet ESP8266 als „keine Fritz!box“

Das Ausrufezeichen besagt, dass dieses Netzwerk keinen Passwortschutz hat. Der ESP kann aber unter der voreingestellten IP 192.168.4.1 im Browser erreicht werden, wenn das NetBook sich mit dem Netzwerk verbunden hat, allerdings ohne Internetzugriff. Mit einem zusätzlichen USB-WiFi-Dongle kann sich das NetBook übriges darüber mit dem Internet verbinden.

Abbildung 2-5: ESP8266 kann kein Internet anbieten als AP

Verantwortlich für dieses Verhalten waren die folgenden Zeilen im Browser.

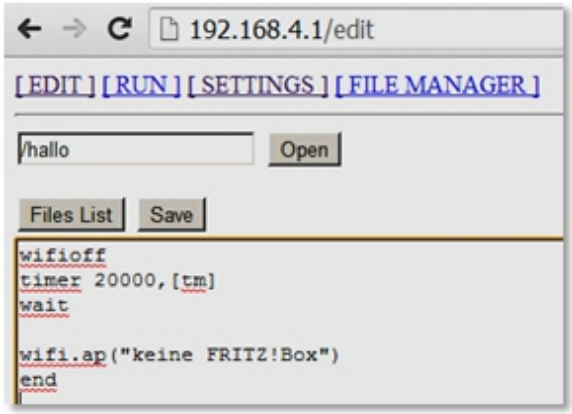


Abbildung 2-6: Programmierung des AP mit Wifi-Anweisungen

Ein Test kann auch so mit nur zwei Zeilen verlaufen:

wifioff

wifi.ap(“keine FRITZ!Box”)

2.4.4 WIFIAPSTA

Soll Access Point und Station-Modus (Router-Verbindung) erforderlich sein, bedient man sich des Befehls *wifiapsta* . Es gibt keine Parameter, da einmal eingestellte AP-Namen und deren Passwörter im ESP-System gespeichert bleiben. Diese Behauptung wird überprüft, indem man aus dem normalen STA-Betrieb heraus WiFi komplett abschaltet und nach einer Weile nur *wifiapsta* bemüht; also ein 3-Zeiler:

wifioff

delay 20000

wifiapsta

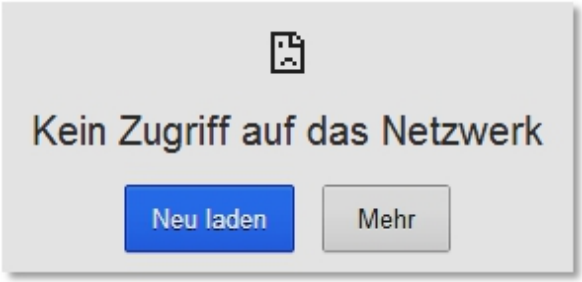


Abbildung 2-7: Chrome-Meldung

In zwanzig Sekunden kann überprüft werden, ob der AP aus der Luft ist. Da dieser Test über den AP lief meldet Google’s Chrome brav entsprechend.

Inzwischen hat sich das NetBook wieder automatisch mit dem Heim-Router verbunden, wodurch die IP des ESP jetzt wieder vom Router vergeben wird. Mit dem Aufruf der IP 192.168.178.30 funktioniert das leider nicht. Die Erreichbarkeit muss also wieder über den eigenen AP erfolgen, da etwas schief gegangen ist. Nach einem Reset läuft nun wieder das *default* -Programm. Um dennoch diese Anweisung zu testen, wird von der Standard-Situation ausgehend, das voreingestellte Programm so geändert, dass die erste Zeile *wifiapsta* ist und dann wird überprüft, ob beide Zugangswege funktionieren.

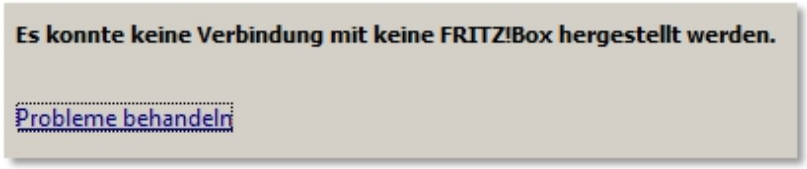


Abbildung 2-8: Windows Meldung

Windows zeigt den AP zwar noch an, aber er ist nicht mehr in der Luft. Die Router Verbindung funktioniert und die Änderung erfolgt auf diesem Weg. Nach dem Verstreichen mehrerer Weilen konnte die Verbindung zum ESP auf beiden Wegen erfolgen.

2.4.5 WiFi Connect und IP

Die normale Verbindung mit einem Router und entsprechendem Internetzugang ist bereits in den Einstellungen von ESPBASIC vorgesehen und bedarf normalerweise keiner Änderung. Sollte es doch vorkommen, dass eine solche Verbindung neu aufgebaut werden soll, erfolgt das über *wifi.connect()*. Dabei sind verschiedene Aufrufparameter möglich:

wifi.connect(SSID)

wifi.connect(SSID, PASSWORT)

und

wifi.connect(SSID, PASSWORT, IP, GATEWAY, NETMASK)

für eine statische IP. Um das Verhalten zu testen, soll im Normalbetrieb am Heim-Router auf den Hot-Spot eines Smartphones umgeschaltet werden. Der Versuch soll mit *wifioff* alle Verbindungen kappen und dann mit Hilfe von *wifi.connect(„Smartphone“,“Passwort“)* mit einem neuen HotSpot Verbindung aufnehmen. Das ginge auch über die Einstellungen und anschließendem Reset, aber das wäre zu einfach. Der erste Entwurf:

wifioff

delay 5000

wifi.connect(“Redmi”,“4e7292711f97”)

wait

Erstaunlicherweise funktioniert das sofort und ohne Probleme. Der ESP ist mit dem neuen Router verbunden. Das NetBook verbindet sich ebenfalls mit dem neuen „Router“ und über die neue IP des ESP kann dieser über den Browser erreicht werden, wenn die IP bekannt wäre… Dieses Smartphone zeigt sie nicht an.

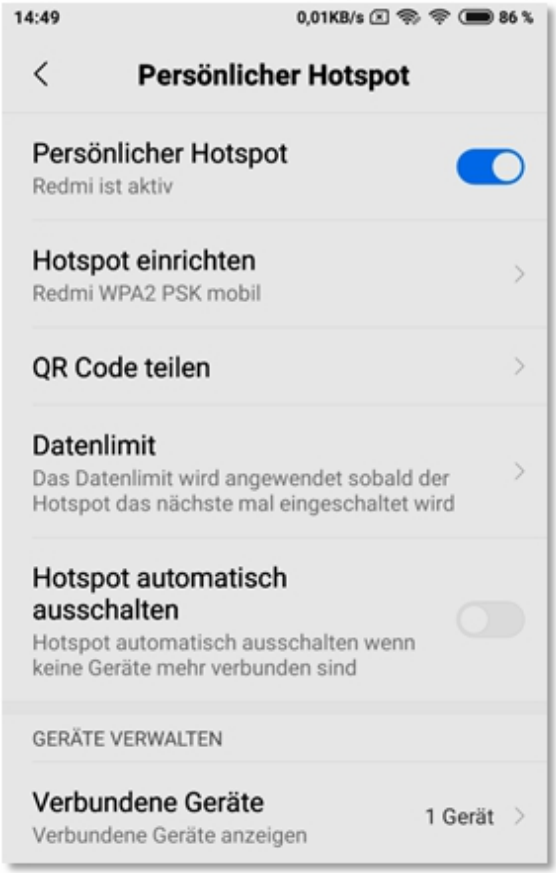


Abbildung 2-9: Smartphone als AP

Es fehlen eine oder zwei Zeilen: *wifiapsta* , damit die neue IP im Browser gelesen werden kann. Wenn das nicht nötig ist, könnte man auch die serielle Ausgabe befragen, aber auf jeden Fall muss ESPBASIC diese neue IP bekannt geben. Die Funktion nennt sich *ip()* und liefert genau das. Ein zweiter Versuch folgt mit der Verbindung über „keine FRITZ!Box“ und eigenem AP und **http://192.168.4.1/** :

wifiapsta

wifi.connect(“Redmi”,“4e7292711f97”)

print ip()

wait

192.168.43.101

Ohne *wifioff* und *delay* verkündet ESPBASIC über den eigenen Hot Spot die neue Router-IP im Browser, sowie seriell, und ist ab sofort auch darunter erreichbar, wenn dasselbe Netzwerk vorliegt.

Abbildung 2-10: ESP über Smartphone AP

Abbildung 2-11: IP im seriellen Monitor (Terminal)

Etwas überraschend tauchen der neue Router und der neue AP auch in den Einstellungen (Settings) auf, so dass nach einem Reset die Konfiguration dauerhaft geändert ist, also nicht nur temporär ein anderer Router benutzt wird. Zugangsdaten könnten so ungewollt verloren gehen. Nach einem *reboot* zeigt sich die neue IP auch in einem seriellen Monitor. Das Ergebnis zu einem anderen Zeitpunkt könnte wie folgt aussehen.

2.5 INTERNET

Ist per WLAN eine Internetverbindung verfügbar, so lassen sich die Web-Funktionen überprüfen.

Lokale IPv4 Abfrage	*ip()*

HTTP Get	*wget()*
JSON	*json()*
Erreichbarkeit	*ping()*
Thingsspeak senden	*sendTS()*
Thingsspeak lesen	*readTS()*
E-Mail initialisieren	*setupMail()*
E-Mail versenden	*email()*

2.5.1 IP UND PING

Die lokale IP kam bereits im vorigen Absatz zum Einsatz. Die Funktion liefert eine Zeichenkette wie z.B. „192.168.178.39“, wenn der ESP mit einem Router verbunden ist. Die IP des ESP als AP ist mit „192.168.4.1“ voreingestellt. Ob eine Internetadresse erreichbar ist überprüft die Funktion *ping()* , die eine 0 oder eine 1 liefert. Die BASIC-Referenz schreibt dazu, dass dies nur in der 4 MB-BASIC-Version funktioniert. Dieser Test lieferte jedoch auch Ergebnisse mit der 2 MB-Version:

```
print version()
print ip()
if ping(“tagesschau.de”) = 1 then
  print “Tagesschau ist online”
else
 print “tagesschau.de ist nicht erreichbar!”
endif
print ping(“192.168.178.30”)
print ping(“hjberndt.de”)
```

ESP Basic 3.0.Alpha 69

192.168.178.39

Tagesschau ist online

0

1

Done…

Bei drei Ping-Abfragen vergeht eine gewisse Zeit bis das Gesamtergebnis im Browser erscheint.

2.5.2 WGET

Ist eine URL erreichbar, so kann deren Inhalte mit der Funktion *wget()* geholt werden. Diese Funktion basiert auf einem http-GET und benutzt als Voreinstellung Port 80. Bei einem anderen Port wird dieser als zweiter Parameter angegeben. Der erste Parameter ist die Adresse (URL), ohne „http://“. Der Test beim Autor von ESPBASIC liefert die Umleitung auf die https-URL.

print wget(“www.esp8266basic.com/”)

Redirecting to https://www.esp8266basic.com/.

Done…

Aufgrund des verfügbaren Speichers im ESP und der möglicherweise großen Datenmenge, funktioniert der Aufruf nur bei kurzen Textseiten. Dafür liegt die folgende kurze Text-Datei zum Test im Netz:

print wget(“www.hjberndt.de/test.txt”)

Diese Datei liegt auf www.hjberndt.de als Test. Vielen Dank.

Done…

Der Inhalt mag sich ändern, aber das Beispiel zeigt, wie HTML-Daten aus dem Netz gelesen werden können.

2.5.3 JSON, OpenWeatherMap

Auf diese Art und Weise sind auch Wetterinformationen aus dem Netz aufrufbar. Ein Dienst, den ESPBASIC besonders unterstützt ist *openweathermap.org* . Dieser Service eignet

sich für kleinere Systeme und kommt bis jetzt ohne *https://* aus. Über die Adresse bzw. URL – wie bei den *msg* - Funktionen – werden die Informationen ausgetauscht. Die Anfrage enthält z.B. eine Stadt und der Dienst liefert die Wetterinformationen in folgender Form (Zeilenumbruch für Drucklayout angepasst):

{“coord”:

{“lon”:8.54,“lat”:52.01},“weather”:

[{“id”:300,“main”:“Drizzle”,“description”:

“light intensity drizzle”,“icon”:“09n”}],

“base”:“stations”,“main”:

{“temp”:279.44,“pressure”:1010,“humidity”:93,

“temp_min”:278.15,“temp_max”:280.15},“visibility”:10000, “wind”:

{“speed”:5.7,“deg”:230},

“clouds”:{“all”:90},“dt”:1547621400,“sys”:

{“type”:1,“id”:1304,“message”:0.1885,“country”:“DE”,“su nrise”:

1547623567,“sunset”:1547653530},“id”:2949186,

“name”:“Bielefeld”,“cod”:200}

Dieses sogenannte JSON-Format unterstützt ESPBASIC mit der *json()*- Funktion. Um den Ort oder die Stadt auszulesen, reicht ein *print json(a, “message.name”)* , wenn die Variable *a* das dargestellte Json-Format enthält. Mit *print val(json(a,“main.temp”))-273.15* erfolgt die Ausgabe der Temperatur, umgerechnet von Kelvin nach Celsius. Ein Listing könnte aussehen wie dieses:

```
memclear
url = “api.openweathermap.org/”
msg = “data/2.5/weather?q=”
ort = “bielefeld,de”
api = “&APPID=5960f9a3ccf9b……..”
a = wget(url & msg & ort & api)
```

let temp = val(json(a,“main.temp”))-273.15

let city = json(a,“message.name”)

print temp

print city

5.85001

Bielefeld

Done…

Die Anfrage erfolgt mit *wget()* und der aus Gründen der Übersicht zusammen gesetzten Anfrage. Die hier unvollständig angegebene *api* -Id erhält man per Mail, nach einer kostenlosen Registrierung bei diesem Dienst. Hinweis: Bei *wget* () darf kein „http://“ vorkommen. *ReadOpenWeather()* soll diesen Dienst auch unterstützen, dies konnte allerdings unter den hiesigen Bedingungen nicht verifiziert werden.

2.5.4 THINGSSPEAK

Die Internetplattform Thingsspeak erlaubt es Messwerte online zu speichern und zu lesen. Außerdem sind Messwertanalysen möglich, auch eine Anbindung an MathLab ist vorgesehen. Mit zwei Kommandos oder Funktionen unterstützt ESPBASIC diese sogenannte Thingsspeak-API.

SendTS(WriteKey, Feld, Inhalt)

ReadTS(ReadKey, Kanal-ID, Feld)

Damit lassen sich Messdaten sehr einfach im Netz speichern und auch wieder lesen. Voraussetzung ist allerdings eine kostenlose Anmeldung mit einer E-Mail-Adresse. Dazu wird dann noch ein Passwort verlangt und nach einer Bestätigung bekommt man Zugang zum System. Wichtig sind die Kanal-ID und die beiden API-Keys zum Lesen und Schreiben, die sich hinter dem Reiter „API Keys“ verstecken. Die Kanal-ID ist zum Lesen erforderlich, zum Senden reicht ein WriteKey.

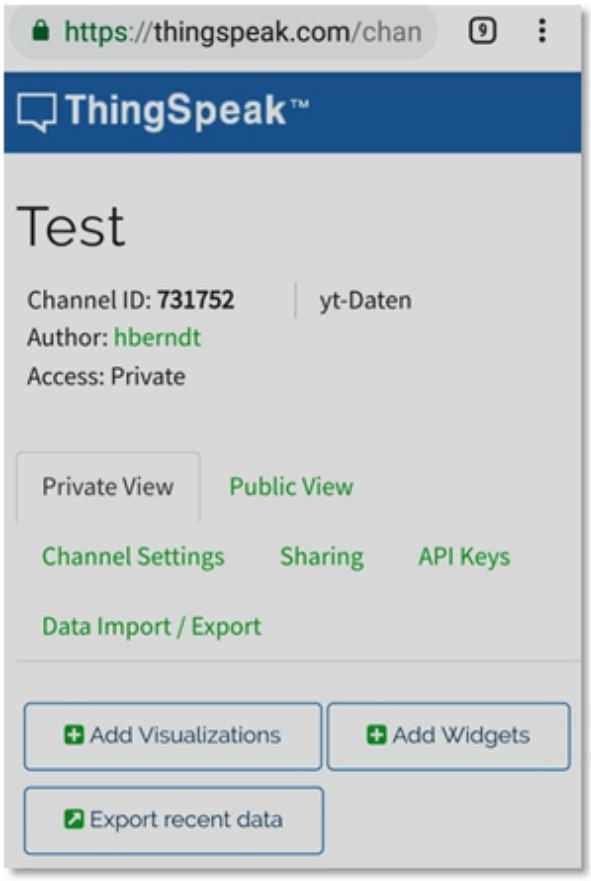


Abbildung 2-12: Nach der Anmeldung bei Thingsspeak

Als Test soll in Kanal „Test“, welcher aus zwei Feldern besteht, in das erste Feld die Zahl 88 übertragen werden. Falls alles fehlerfrei verläuft, erfolgt das Lesen dieses Wertes und die Ausgabe. Die Ansicht der 88 bei Thingsspeak schließt den Test ab.

W$ = “5J7ZGGD4M2H55QGJ”

R$ = “AG6RNN3DAS12K0Q2”

C$ = “731752”

Sendts(W$,“1”,“88”)

Print readts(R$,C$,“1”)

88

Done…

Die drei Schlüssel wurden verändert und sind so nicht gültig; hier müssen die eigenen Keys eingefügt werden. In *W$* steht der Key zum Senden, in *R$* , der Key zum Lesen und die

Kanal-ID ist in *C$* gespeichert. Alle Werte sind Zeichenketten, das gilt auch für Messdaten.

Wie man den Daten entnehmen kann, dauerte es mehr als zehn Minuten, bis das erste Datum nach der Anmeldung erfolgreich angekommen ist. Das lag in erster Linie daran, dass das Ganze am Smartphone in einem Chrome-Browser ablief, der beim Kopieren und Tab-Wechsel manchmal etwas empfindlich auf Berührungen reagiert. Auch gab es Parameter-Verwechslungen beim Lese-Aufruf.

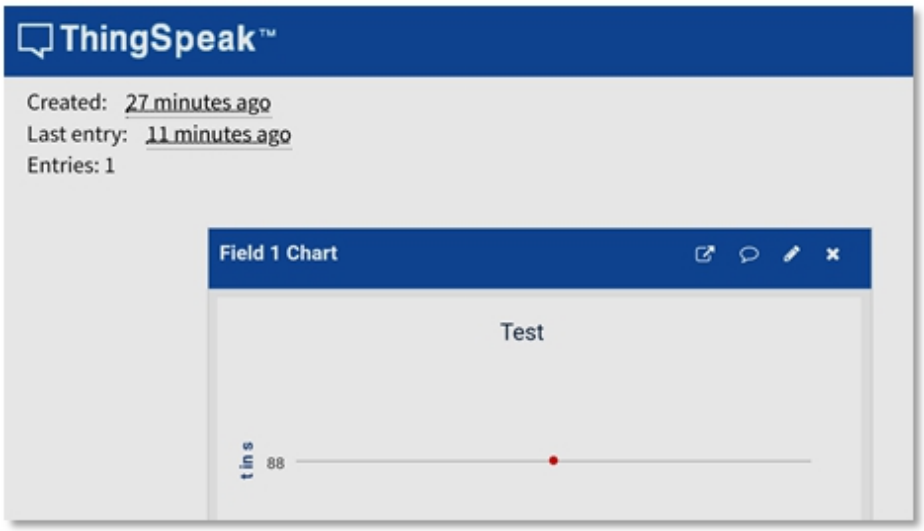


Abbildung 2-13: Erster Messpunkt „88“ in Thingsspeak

2.5.5 MAIL

ESPBASIC stellt zwei Mail-Funktionen zur Verfügung. Der Konfigurationsaufruf *SetupMail* benötigt die Parameter Server, Port, Benutzername und Passwort. Danach kann mit *eMail* und den Parametern Anschrift, Absender, Betreff, sowie dem eigentlichen Text der Botschaft, die Post auf den Weg gebracht werden.

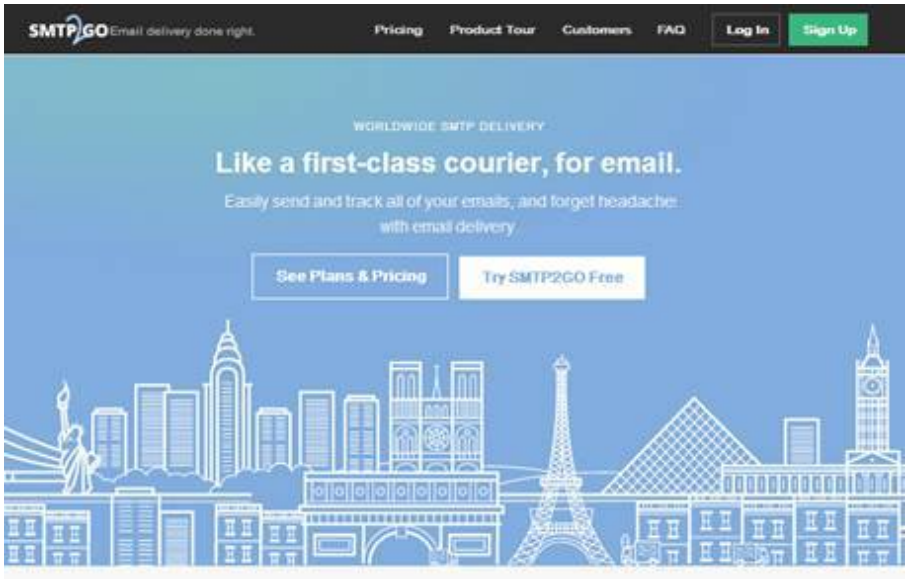


Abbildung 2-14:Homepage von SMTP2GO

Mit nur zwei Zeilen kann man in ESPBASIC eine E-Mail versenden. Dazu ist natürlich die Internetverbindung über WiFi von weiter oben erforderlich.

setupemail “mail.smtp2go.com”, 2525, “username”, “mailpassword”

email “empfänger@mail.com”,“username@smtp2go.com”, “Betreff”,“Inhalt”

Wie in der ESPBASIC-Referenz beschrieben, bietet sich ein kostenfreier Probe-Account bei *smpt2go.com* an, da bei den meisten Servern vorhandener Accounts dieser einfache Aufruf nicht funktioniert.

Die dortige Anmeldung erfordert einen neuen Usernamen und eine vorhandene E-Mail-Adresse, sowie ein neues Passwort zum Einloggen auf der Seite von *smpt2go.com* . Nach der Anmeldung bekommt man über die angegebene E-Mail-Adresse eine Bestätigung mit der Aufforderung der Verifizierung. Dabei landet man als angemeldeter Nutzer auf einer Seite, auf der man ein E-Mail-Password erhält, was entsprechend eigener Vorstellungen geändert werden kann. Zum Schluss öffnet sich das Dashboard mit den Aktivitäten.

Damit ist der Vorgang abgeschlossen. Die angegebene E-Mail-Adresse ist gleichzeitig das Ziel der weitergeleiteten Mails. Angenommen der LoginName sei *otto2go* und es gäbe einen bestehenden Account *otto@otto.de* , wohin auch die Mails des ESP verschickt werden sollen, der Username bei smpt2go sei ebenfalls *otto@otto.de* und das Passwort für Login und E-Mail

ist *otto2000* , dann könnte eine Testmail vom ESP mit den Worten “Hallo Welt” mit folgenden Zeilen versendet werden:

setupemail “mail.smtp2go.com”, 2525, “otto@otto.de”, “otto2000”

email “otto@otto.de”,“otto2go@smtp2go.com”, “Testmail”,“Hallo Welt”

Der Probe-Account gestattet nur 25 E-Mails pro Stunde und 1000 E-Mails pro Monat. Für Testzwecke und andere Anwendungen ein funktionierendes Verfahren, womit sich ganze Messprotokolle verschicken lassen. Während einer Nacht sendeten der ESP8266 und das unten angegebenen BASIC-Listing bis zum frühen Morgen Messdaten per Mail.

Abbildung 2-15: ESPBASIC verschickt Mails

Das folgende Beispiel benutzt bereits die eingebaute DHT-Bibliothek des Temperatur- und Luftfeuchte Sensors, die weiter unten einzeln beschrieben wird.

```
dht.setup(11, 2)
crlf = chr(13) & chr(10)
timer 3600000,[messen]
‘wait
[messen]
t = DHT.TEMP()
h = DHT.HUM()
i = DHT.HEATINDEX()
l = io(ai)
msg = “Temperatur: ” & t & crlf & “Gefuehlt: ” & i & crlf
```

```
msg = msg & “Luftfeuchte: ” & h & crlf & “Helligkeit: ” & l
print msg
setupemail “mail.smtp2go.com”, 2525, “otto@otto.de”, “otto2000”
email “otto@otto.de”,“otto2go@smtp2go.com”, ““ESP8266 at ” & time(), msg
wait
```

Nach der Initialisierung des Sensors an Pin 2 wird ein *Timer* auf 60 Minuten gestellt und sofort aufgerufen, wenn das *wait* auskommentiert ist. Danach folgen die Messung und die Erstellung der Meldung in *msg* . Mit der UTC-Zeit im Betreff wird die elektronische Post auf den Weg gebracht.

Mit diesem Verfahren, dem Autostart von */default.bas* und dem *Sleep* -Befehl für den ESP sollten auch Langzeitmessungen mit geringem Energiebedarf möglich sein. Dabei schläft der ESP8266 zum Beispiel drei Stunden bei einem Strombedarf von wenigen Mikroampere, wacht auf, verbindet sich mit der WiFi-Station, startet das BASIC-Programm, schickt die Messwerte ab und schläft weiter. Dank ESPBASIC bleibt die Sache erfreulich einfach und übersichtlich.

2.6 HTTP-SERVER

Messen und Steuern über IP/TCP mit dem ESP8266 ist ein Thema in [3]. Dort fehlt die Kommunikation mit Programmen wie z.B. Word und Excel (VBA), da diese Sockets (TCP/IP)-Kanäle nicht unterstützen. Eine Leseranfrage bezüglich dieses Problems führte zu einer Nachreichung auf ***http://hjberndt.de/soft/esp2excelbas.html*** . VBA und andere Programmierumgebungen, wie z.B. RFO-BASIC auf Android kennen die HTTP-GET Funktion und können so auf Seiteninhalte zugreifen.

Wenn nun die fremde Applikation (Excel/VBA/RFO) den Seitentext abrufen kann, so muss nur noch ein Server entsprechende Messwerte als kurze Texte bereitstellen. ESPBASIC selber benutzt einen kleinen HTTP-Server, um z. B. seine Programmierumgebung im Browser darzustellen. Mit „**http://192.168.178.39/edit** “ in der Adresszeile editiert man ein Programm direkt über die Tastatur. Das ESPBASIC-System verzweigt also anhand des dem Slash „/“ folgenden Schlüsselwortes. Ändert man in der Zeile das „edit“ zu „run“ und betätigt die Eingabetaste, so ist das identisch mit einem Klick auf RUN. Das Schlüsselwort „msg“ benutzt das BASIC-System um dem BASIC-Programm die vom Anwender angegebenen Zeichen nach dem Wort „msg“ zu übermitteln. Der Programmierer kann dann entsprechend selbst in seinem Programm Verzweigungen einbauen. Der Befehlsvorrat zu diesem sogenannten *Web-Msg-Handler* Verfahren ist:

msgBranch

---

msgGet

---

msgReturn

Ist ein ESP am Router mit der IP 192.168.178.39 verbunden, und lautet der Text in der Adresszeile *http://192.168.178.39/msg?pin=15,* dann erhält man den Wert 15 wie folgt:

msgbranch [mybranch]

wait

[mybranch]

print msgget(“pin”)

wait

Dazu wird das Programm gespeichert und gestartet. BASIC wartet nun auf das Eintreffen des Schlüsselwortes *msg* . Wird in einem Browser der obige Text eingegeben, so springt ESPBASIC an die durch *msgbranch* angegebene Stelle und das Programm erhält den Wert über *msgget* . Mit *msgreturn* erfolgt eine Rückmeldung im Browser, wie zum Beispiel hier:

Abbildung 2-16: Kommunikation über die Adresszeile mit msgget

msgbranch [mybranch]

wait

[mybranch]

pin = msgget(“pin”)

msgreturn “Eingabe: ” & pin

wait

Die Besonderheit von *msgreturn* ist, dass die vorige Ausgabe wieder gelöscht wird. Dass ist nützlich, wenn verschiedene Messwerte an derselben Stelle gelesen werden müssen, wie im nächsten Beispiel. Soll also über die Adresszeile gemessen und gesteuert werden, so muss der Rückgabewert des Pins noch in eine numerische Variable gewandelt werden, um so eine LED an Pin 2 zu schalten, ein weiterer Parameter legt den Zustand (*wert*) der LED fest. Umgekehrt kann der Helligkeitswert des LDR am Witty Cloud mit der Analogabfrage mittels *msgreturn* geliefert werden. Die Ein- und Ausgaberoutinen sind weiter oben erläutert. Diese Anwendung schaltet den angegebenen Pin wie gewünscht und liefert den tatsächlichen Status im Browser zurück.

msgbranch [mybranch]

wait

[mybranch]

```
pin = msgget(“pin”)
wert = msgget(“wert”)
io(po,val(pin),val(wert))
ist = io(laststat,val(pin))
msgreturn “pin ” & pin & ” ist ” & ist
wait
```

Die Analogabfrage als Einzelprogramm kommt mit wenigen Angaben aus, da es nur einen Eingang gibt, der mit *ai* abgefragt wird. Der Aufruf erfolgt mit *http://192.168.178.39/msg?,* das ESPBASIC-Programm dazu ist auch sehr, sehr kurz.

```
msgbranch [mybranch]
wait
[mybranch]
msgreturn io(ai)
wait
```

https://www.esp8266basic.com/msg-url-advanced.html

zeigt ein weiteres Beispiel zu dieser Technik, die hier abgewandelt benutzt wird.

Nach diesem System kann nun auch in Word und Excel mit dem ESP8266 direkt über ESPBASIC gemessen und gesteuert werden. Beispiele dazu sind im Abschnitt Anwendungen zu finden.

2.7 SERIELL ZU WIFI

Eine Grundeigenschaft des ESP8266 ist die Verbindung der beiden seriellen Übertragungsarten TCP/IP und RX/TX. Seine RX/TX Leitungen mit 3,3 Volt-Pegel können Informationen über WiFi/WLan übernehmen und auch umgekehrt.

2.7.1 SERIELL ZU TCP/IP – CLIENT

ESPBASIC selber agiert als HTTP-TCP/IP-Server und der Anwender kann über die *msg* -Routinen (siehe oben) am Server teilnehmen. Als Client benutzt ESPBASIC dazu die sogenannten Telnet-Routinen, die einen Telnet-Client bereitstellen.

Telnet.Client.Connect()
Telnet.Client.Available()
Telnet.Client.Read.Chr()
Telnet.Client.Read.Str()
Telnet.Client.Write()
Telnet.Client.Stop()
TelnetBranch

Auffallend ist die Ähnlichkeit zu den Seriell-Routinen aus Abschnitt 2, wodurch alles sehr übersichtlich wird. Auch die Programmstruktur ähnelt sich dadurch und muss nicht erneut im Detail erläutert werden. Den Bezug stellt die ganz zu Beginn in Abschnitt 1 schon benutzte *INPUT* -Methode im Browser mit einer *Textbox* aus dem Abschnitt Web-Interface dar.

Ein serielles Gerät liefert über die RX/TX-Leitungen Kommandos und Messwerte an einen ESP8266, der diesen

Datenstrom per TCP/IP über das Netz einem anderen Server weiter leiten soll. Der Einfachheit wegen soll das serielle Gerät ein Smartphone sein, wie es mit einer Terminal-App im Abschnitt 2 bereits benutz wurde. Auf der anderen Seite soll ein Windows-Rechner als Telnet-Server dienen. Der ESP8266 ist lediglich ein Mittler zwischen den seriellen Welten.

Ein PC lauscht am Port 23 und wartet auf ankommende Daten. Die Freeware *RealTerm* für Windows wird in [3] beschrieben und benutzt, um eine ähnliche Verbindung aufzubauen. Als Portnummer könnten niedrige Werte vom System gesperrt sein, dann muss auf höhere Ports ausgewichen werden. Dieser (Windows-)Server wartet also nach dem Start auf Port 23 auf einen Client, hier das ESP-Programm. Das Smartphone mit seiner App verbindet sich über OTG-Kabel mit dem ESP und kann so serielle Daten vom Handy übermitteln. Die einzelnen Schritte gestalten sich dann wie folgt, wenn BASIC bereits im ESP läuft

- BASIC-Programm in der ESP8266BASIC-Editor des Browsers übertragen
- ESP8266 über USB -(Host)-Kabel/Phone/Serial-Terminal verbinden
- RealTerm aufrufen und Telnet-Server starten, da sonst keine Verbindung
- BASIC-Programm speichern (Save) und starten (Run)

Das BASIC-Programm benötigt die IP vom Server. Um ohne Router zu arbeiten, verbindet sich der PC mit dem vom ESP8266 aufgespannten Netz über seinen AP.

Über den Taskmanager oder *ipconfig* im Kommandofenster erhält man bei Verbindung mit dem Hot Spot des ESP ohne Router:

Windows-IP-Konfiguration

Drahtlos-LAN-Adapter Drahtlosnetzwerkverbindung:

Verbindungsspezifisches DNS-Suffix:

Verbindungslokale
IPv6-Adresse . : fe80::e921:27a9:c984:f78a%14

IPv4-Adresse … … … . : 192.168.4.2

Subnetzmaske … … … . : 255.255.255.0

Standardgateway … … … : 192.168.4.1

Also lautet das Listing

```
textbox T$
textbox S$
print “Hi. TCP/IP2Serial mit esp8266Basic”
if telnet.client.connect(“192.168.4.2”,23)>0 then
 telnet.client.write(“Hallo. Hier spricht ESPBASIC!”)
 Telnetbranch [clientread]
 serialbranch [serialread]
else
 print “Keine Verbindung.”
 end
endif
wait
[clientread]
T$ = telnet.client.read.str()
serialprint T$
wait
[serialread]
serialinput S$
telnet.client.write(S$)
return
[ende]
telnet.client.stop()
end
```

Die zwei *TextBoxes* sind die Anzeigen im TCP- und seriellem Modus. Das Programm nutzt die Ereignissteuerung und verzichtet damit auf ständiges Abfragen der Verfügbarkeit von Daten der jeweiligen Schnittstelle. Zu beachten ist, dass in der hier verwendeten Version 3.3 der serielle Branch “*SerialRead* ” mit *Return* endet -, der Socket-Branch “*ClientRead* ” aber mit einem *wait* abgeschlossen ist. Immer wenn Daten anliegen, werden diese unverändert an die andere Schnittstelle weiter gereicht. Schließlich wird mit etwa zwei Zeilen die

Server-Verbindung mit der entsprechenden IP aufgebaut und am Ende wieder geschlossen.

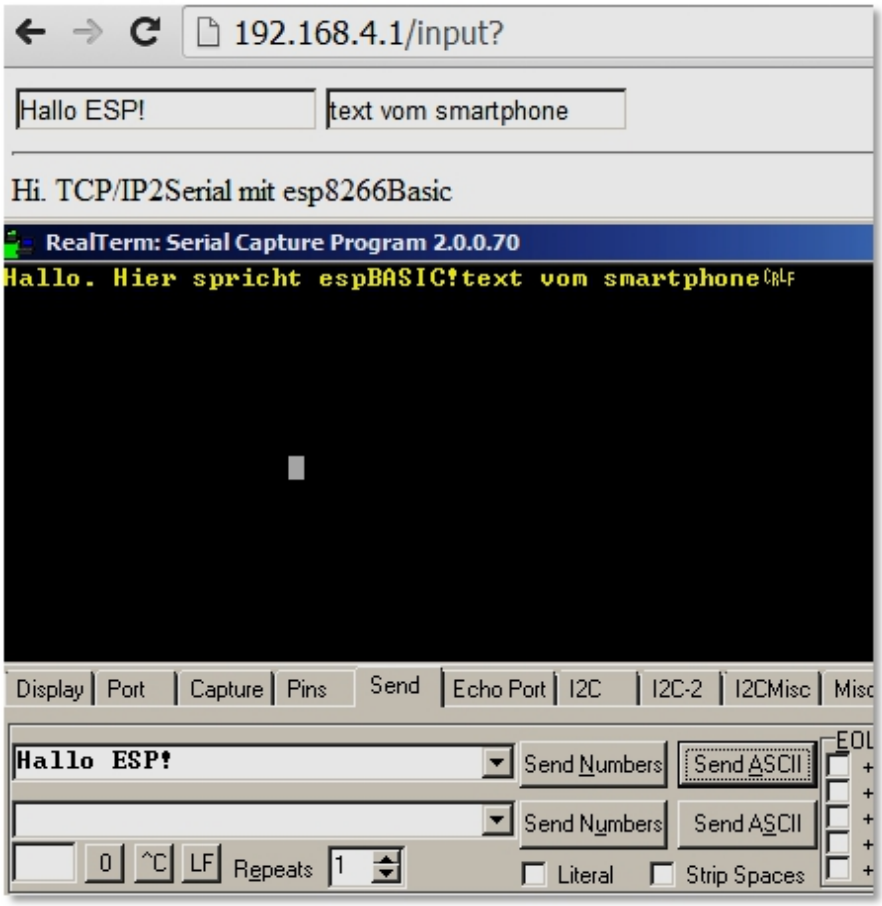


Abbildung 2-17: RealTermfür Windows und ESPBASIC

Der PC zeigt im Vordergrund RealTerm als Telnet-Server mit einem gesendeten Gruß, der im darunter liegenden Browser in der linken *TextBox* erscheint. Die rechte *TextBox* zeigt die seriellen Daten vom Smartphone, oder eines anderen über RX/TX des ESP verbundenen Geräts. Das BASIC-Programm ist nur der Vermittler und verhält sich völlig passiv. Die *Textboxes* zeigen nur eine Kopie der übertragenen Daten und dienen nicht der Eingabe.

Somit erfüllt der ESP-Baustein seine ureigene Aufgabe: TCP/IP-Serial-Konverter.

2.7.2 Seriell zu UDP/IP - Server

Einen anderen Weg in ESPBASIC Daten der seriellen Schnittstelle als Pakete ins Netz zu schicken bieten die UDP-Routinen. Es werden 8 Befehle und Funktionen zur Verfügung

gestellt, wobei der *udpwriteMuliCast* am Ende dieses Abschnitts anders als in der Referenz gelöst wird.

udpbegin
udpstop
udpwrite
udpread()
udpremote()
udpreply
udpbranch
udpbeginMultiCast
udpwriteMultiCast

Mit *udpbegin* wird ein udp-Server gestartet, mit *udpstop* wieder angehalten. Geschrieben wird mit *udpwrite* . Als Beispiel soll zunächst nur die Konnektivität überprüft werden, indem ein Server aktiviert wird und ein UDP-Client etwas empfängt. Der ESP-UDP-Server lauscht auf Port 8888 des ESP. Auf dem Android-Phone läuft das werbefinanzierte „UDP Terminal 1.4 von mightyIT“ aus dem Playstore.

```
port = 8888
udpbegin port
print “Server started at ” & port
udpbranch [udp.received]
wait
[udp.received]
print “udp: ” & udpread()
return
```

Server started at 8888

---

udp: UDP Test

---

udp: UDP Test

---

udp: moin

---

udp: UDP Test

Um serielle Daten über UDP zu verschicken, wird der ESP in diesem Beispiel als autarker AP betrieben; es soll ein zugängliches Exemplar sein, da das Board über OTG am USB-Anschluss eines Android-Phones betrieben werden soll, um wiederum serielle Daten per Terminal dem ESP zukommen zu lassen. Auf demselben Phone läuft auch das UDP-Terminal im Split-Screen. Das Programm-Listing ist eine Modifikation aus dem Abschnitt *2.1 Serielle Kommandos und Funktionen* . Die einzelnen Darsteller und deren IP sind:

ESP8266 192.168.178.1 als Seriell zu UDP Konverter

Android 192.168.178.3 als Serieller Sender und Packet Empfänger

Windows 192.168.178.2 als Programmier-Browser

```
port = 8888
udpbegin port
serialbranch [rx]
print “Server startet at ” & port
udpbranch [udp]
wait
[udp]
a = udpread()
print “udp: ” & a
udpreply “OK, :” & a
return
[rx]
serialinput b
print “RX: ” & b
udpwrite “192.168.4.3”,8080,b
return
```

Das Testergebnis auf dem Sende- und Empfangsgerät:

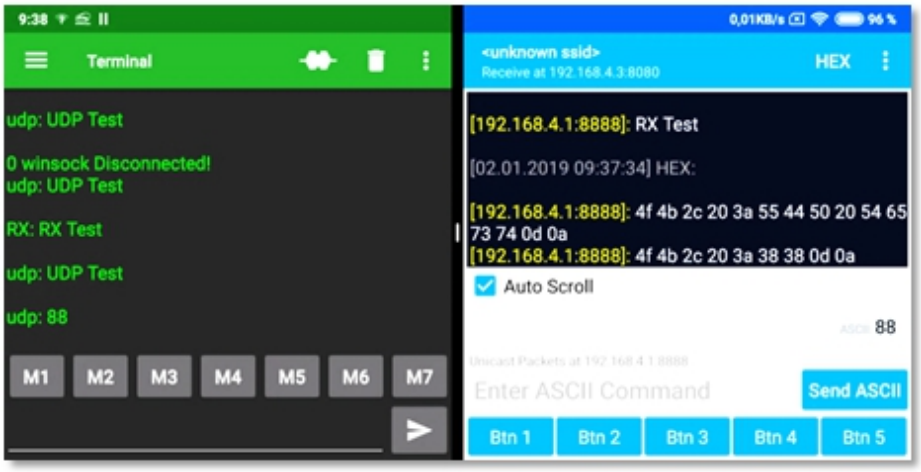


Abbildung 2-18: Seriell und UDP nebeneinander

Auch hexadezimale Darstellungen sind möglich, wobei das Leerzeichen mit dem Wert 20 (32) und die beiden Bytes 0d oa für CrLf stehen. Der Rest sind die entsprechenden ASCII-Werte der Zeichen. Die Einstellungen für diese Bedingungen waren wie folgt.

Abbildung 2-19: Einstellungen im UDP-Terminal

Mit *udpwrite* kann man an eine bestimmte IP mit angegebenem Port Meldungen versenden. Der Aufruf wäre *udpwrite "192.168.4.5", 1234, "Hallo"* . Mit *udpremote()* erhält man die IP und den Port der empfangenen Meldung in der Form „192.168.4.3:8085".

An mehrere Geräte per UDP senden

Mit *udpMultiCast* kann mit verschiedenen Geräten in einem IP-Bereich kommuniziert werden. Der IPv4-Bereich ist mit 224.0.0.0 bis 239.255.255.255 angegeben, sagt die Referenz zu ESPBASIC. Durch Versuche konnten jedoch auch durch entsprechende Schleifen mit *udpwrite()* mehrere Geräte in anderen Bereichen mit Messdaten versorgt werden.

Ein Beispiel, welches bei der Entwicklung einer Anwendung im Abschnitt 4 Verwendung findet, ist das Versenden von Messdaten an mehrere Geräte. Dort wird der relative Fehler einer Messung per UDP verschickt. Mit

```
udpbegin 8080
udpbranch [udp]
```

erfolgt die Initialisierung mit Port 8080 und dem Sprungziel *[udp]* für eingehende Daten (nicht unbedingt erforderlich). An der gewünschten Stelle in einer Messroutine erfolgt dann die Verteilung nach dem Gießkannen-Prinzip:

```
temp = replace(str(frel),".",",")
for cl = 20 to 34
 u$ = "192.168.178." & str(cl)
 udpwrite u$, 8080, temp
next cl
```

Der umgewandelte numerischen Messwert *frel* erfährt eine Punkt zu Komma Ersetzung, um die Zeichenkette *temp* so zu verschicken, dass der Zahlenwert wie gewünscht ankommt. Diese Zeichenkette wird über Port 8080 an die Clients *cl* 192.168.178.20 bis 192.168.178.34 verschickt.

Falls ein zuhörender Client etwas zurück meldet, wird das in diesem Beispiel im *branch [udp]* eingefangen, verworfen und mit dem Originalmesswert quittiert:

```
[udp]
a = udpread()
udpreply frel
return
```

Unkompliziert und kurz, wie so oft in ESPBASIC.

2.8 GRAFIKAUSGABEN

ESPBASIC kann ein Browser Fenster als Ausgabekanal benutzen, wie bisher mit der Print-Anweisung schon öfter geschehen. Zusätzlich können in diesem Fenster zur Laufzeit graphische Elemente, wie Rechtecke, Kreise und Linien Verwendung finden. Als Interface werden in diesem Zusammenhang Ein- und Ausgabeelemente bezeichnet, die HTML5 auf Web-Seiten zur Verfügung stellt und hier die Interaktion mit dem Anwender erlauben.

2.8.1 GRAFIK GRUNDELEMENTE

Zu den Befehlen und Funktionen der Web-Grafiken zählen:

Farbpalette	Color Palette
Grafikelement erstellen	*Graphics*
Grafikelement löschen	*Gcls*
Linie zeichnen	*Line*
Kreis zeichnen	*Circle*
Ellipse zeichen	*Ellipse*
Rechteck zeichnen	*Rect*
Positionierter Text	*Text*

Hinweis: *Diese graphischen Elemente stehen erst ab einer 2MB-BASIC-Version zur Verfügung!*

Die *Farbpalette* umfasst 16 Farbvariationen von 0 für Schwarz bis 15 für Weiß als numerische Werte. Mit der Anweisung Graphics wird eine Zeichenfläche erstellt mit der angegebenen Höhe und Breite in Pixel-Einheiten. *Gcls* löscht diesen

Grafikspeicher. Die weiteren Anweisungen sind eher selbsterklärend, ihre Syntax lautet jeweils:

Line x1, y1, x2, y2, Farbe

Circle x, y, Radius, Farbe

Ellipse x, y, Radiusx, Radiusy, Farbe

Rect x, y, Breite, Höhe , Farbe

Text x, y, Zeichenkette, Winkel, Farbe

Farb-Palette

Ein Beispiel zeigt auf einer 640 x 480 Pixel großen Zeichenfläche die Farbpalette.

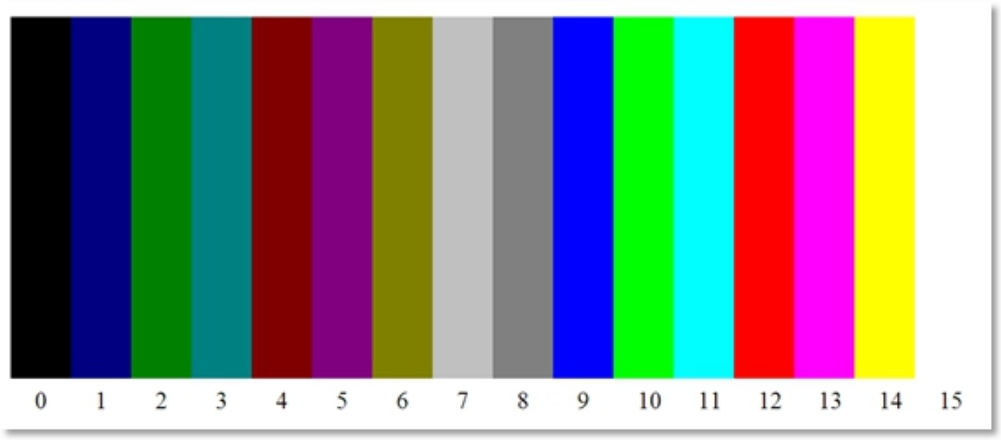


Abbildung 2-20: Farbpalette von und mit ESPBASIC

```
graphics 640, 480
for farbe = 0 to 15
 rect 640/16*farbe+0 ,0, 640/16, 240, farbe
 text 640/16*farbe+16, 260, str(farbe)
next i
```

Weitere kurze Beispiele zur Grafikausgabe:

```
w = 240
h = 240
graphics w, h
text w*3/6+2, h*1/2+2, “ESP-BASIC”, 45, 7
text w*3/6+0, h*1/2+0, “ESP-BASIC”, 45, 0
for x = 0 to w-1 step 10
 line 0,0,x,h-1, 8
```

next i

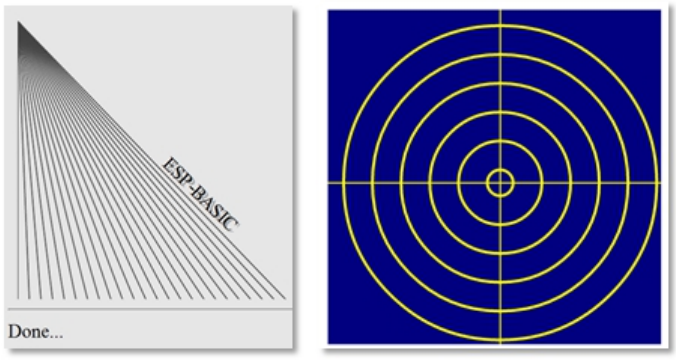


Abbildung 2-21: Grafik mit ESPBASIC

```
w = 240
h = 240
graphics w, h
rect 0,0,w-1,h-1,1
for r = h/2-10 to 10 step -20
 circle w/2,h/2,r, 14
 circle w/2,h/2,r-2,1
next i
line w/2,0,w/2,h-1,14
line 0,h/2,w-1,h/2,14
```

2.8.2 GRAFISCHE OBERFLÄCHE, WEB-INTERFACE

Die ESPBASIC-Programmieroberfläche ist ein Browserfenster mit verschiedenen HTML-Elementen, die über JavaScript eingebunden sind. Dem Programmierer stehen solche Elemente ebenfalls zur Verfügung, so dass ansehnliche Benutzeroberflächen konstruiert werden können, wie sie von Webseiten bekannt sind. ESPBASIC nennt das ein GUI (GraphicalUserInterface). Bei zeit- und speicherkritischen Anwendungen ist dieser Komfort jedoch mit *guioff* abschaltbar. Dann funktionieren *print* , *wprint* , *button* , usw. nicht mehr. Dies gilt auch für Web-Sockets. Der Befehl *guion* aktiviert die Elemente wieder. Die folgende Übersicht zeigt die vielfältichen Möglichkeiten des Web-Interfaces:

Wprint/Html
Button
Textbox
Meter
Slider
Dropdown
Listbox
Wait
Image
Imagebutton
Cls
Passwordbox
Onload
HtmlVar()
HtmlID()
GuiON

GuiOFF
CSS
CSSID
CSSCLASS
JAVASCRIPT
JSCALL

In der Reihenfolge einer subjektiven Relevanz bezogen auf das Thema dieses Buches, sollen kleine Beispiele die Anwendung der Elemente verdeutlichen.

2.8.3 PRINT, WPRINT, HTML

Der Print-Befehl von ESPBASIC ruft intern zwei andere Befehle auf. Die Ausgabe erscheint sowohl als *HTML* auf der Browserseite mit *Wprint* als auch am seriellen Ausgang wie bei *SerialPrint* . Soll die Ausgabe nur im Browser erscheinen, so erfolgt das über *Wprint* bzw. *Html* . Mit dieser Ausgabe können auch HTML-Elemente „gedruckt“ werden. Ein Beispiel:

```
html |<font face="Helvetica"><hr noshade size="1">|
wprint "Oben und unten eine horizontale Linie"
html "<hr><h3>"
html |"Anf&uumlhrungszeichen/Umlaute in HTML!"<br>|
button " ENDE ",[ok]
html "</font>"
html "Warte auf ENDE…"
wait
[ok]
end
```

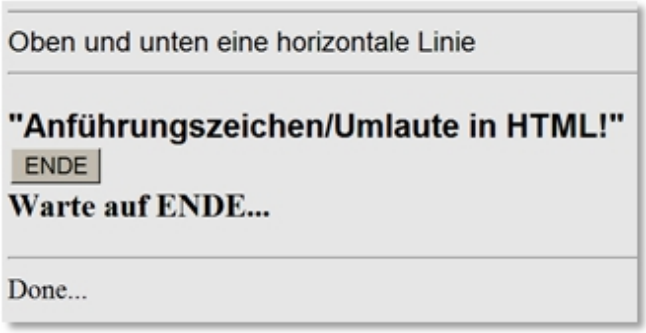


Abbildung 2-22: Web-Grafik mit ESPBASIC

Die Kurzform von *Wprint* ist *Html* , beide sind identisch. Üblicherweise sind konstante Zeichenketten in BASIC von Anführungszeichen umgeben. Da aber bei HTML-Code auch diese Zeichen benötigt werden, kennt ESPBASIC das „|“ als ein Alternative.

Slider, TextBox, Meter

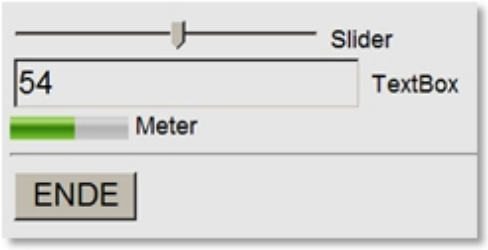


Abbildung 2-23: HTML5-Elemente und Web GUI

```
html |<font face=“Helvetica” size = “1”>|
slider x, 0,100
html ” Slider <br>”
textbox x
html ” TextBox<br>”
meter x, 0,100
html ” Meter <hr>”
button ” ENDE “,[ok]
html “</font>”
wait
[ok]
end
```

Alle Elemente sind hier mit derselben Variablen *x* verknüpft, wodurch alle Elemente denselben Wert anzeigen. Auch

Eingaben in der *TextBox* ändern *meter* und *slider* ! Dieses Beispiel zeigt wie einfach die Verknüpfung von Element und Variable ist. Natürlich können die Elemente auch alle mit verschiedenen Variablen verknüpft sein. Eine Messwertanzeige für den LDR am Witty Cloud an A0 mit 100 ms-Intervall und einem *meter* ist schnell und kurz programmiert:

```
timer 100,[tm]
meter ldr, 0, 1000
wait
[tm]
ldr = io(ai)
wait
```

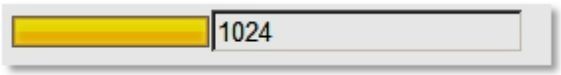


Abbildung 2-24: Meter und Textbox

Fügt man unter der *meter* -Zeile noch ein *TextBox ldr* ein, erscheint dort, bei Verknüpfung mit der Variablen *ldr* , der entsprechende numerische Wert. Da der Bereich des Meters zu gering angegeben ist, wechselt das HTML5-Element bei Überschreitung der Grenzen die Farbe – zumindest in Opera.

2.8.4 DROPDOWN UND LISTBOX

Dropdowns und Listboxen dienen der Auswahl. Die Programmierung ist hier auch denkbar unkompliziert:

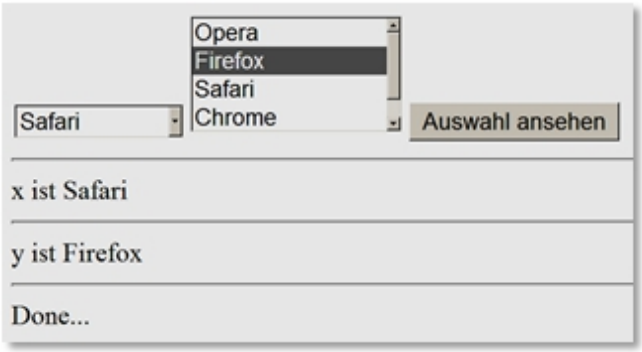


Abbildung 2-25: Oberfläche mit Auswahl mittels Web-Elementen

```
x = “Firefox”
y = “Chrome”
dropdown x, “Opera,Firefox,Safari,Keine Ahnung”
```

```
listbox y, “Opera,Firefox,Safari,Chrome,Internet Explorer 4”, 3
button “Auswahl ansehen”,[ok]
wait
[ok]
html “<hr>x ist ” & x
html “<hr>y ist ” & y
end
```

2.8.5 CSS, CSSID, CSSCLASS

Mit den Mitteln von CSS lassen sich HTML-Elemente den eigenen Wünschen anpassen.

Der Autor von ESPBASIC gibt ein Beispiel an, welches den CSS-Teil in einer eigenen Datei „test.css“ speichert, und zwar so, dass es per Dateimanager hochgeladen wird. Dazu kopiert man den CSS-Text und fügt ihn z. B. in ein neues Textdokument ein. Nach Speicherung und Umbenennung in „test.css“ kann dieser Text mit „Datei auswählen“ am PC oder Smartphone gewählt werden. Mit „Upload“ landet diese Datei „test.css“ im Dateisystem des ESP8266 als „/uploads/test.css“:

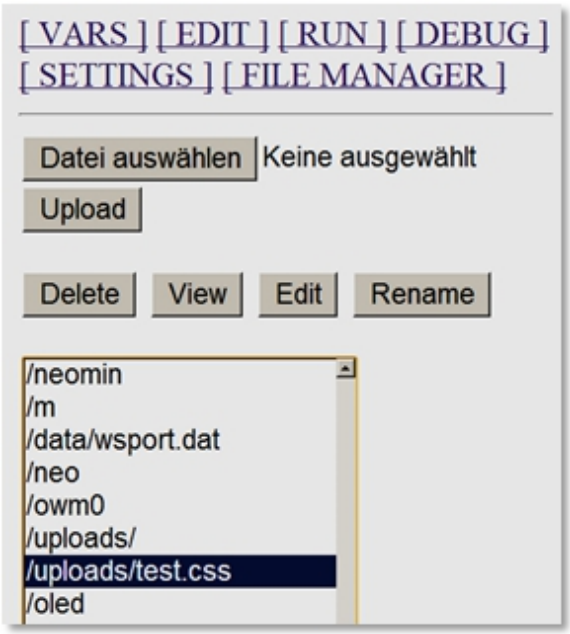


Abbildung 2-26: Hochgeladene Datei im Dateimanager von ESPBASIC

Mit der Zeile *css „test.css“* wird diese Datei dann eingebunden, die Pfadangabe *„/uploads/“* entfällt, da ESPBASIC solche und andere Dateien in diesem Verzeichnis sucht.

Das eigentliche ESPBASIC-Programm ist dadurch kurz. Durch Voranstellen eines Hochkommas kann die erste Zeile zu Testzwecken auskommentiert werden. Dadurch lassen sich die Unterschiede zwischen Original und Änderung beobachten. Das Originalbeispiel von ESP8266BASIC.com:

css “test.css”

textbox test

dropdown test2 ,“1,2,3,4,5”

button “Click me”, [branch]

wait

[branch]

print “You clicked me”

wait

Den Inhalt der externen Datei findet man ebenfalls auf
https://www.esp8266basic.com/css-example.html
und sei hier nur der Vollständigkeit halber nochmal aufgeführt.

CSS CODE:

```
button, input[type=“button”], input[type=“submit”] {
    border-top: 1px solid #96d1f8;
    background: #000000;
    background: -webkit-gradient(linear, left top, left bottom, from(#0099ff), to(#000000));
    background: -webkit-linear-gradient(top, #0099ff, #000000);
    background: -moz-linear-gradient(top, #0099ff, #000000);
    background: -ms-linear-gradient(top, #0099ff, #000000);
    background: -o-linear-gradient(top, #0099ff, #000000);
     padding: 20px 40px;
    -webkit-border-radius: 31px;
    -moz-border-radius: 31px;
    border-radius: 31px;
     -webkit-box-shadow: rgba(0,0,0,1) 0 1px 0;
    -moz-box-shadow: rgba(0,0,0,1) 0 1px 0;
    box-shadow: rgba(0,0,0,1) 0 1px 0;
    text-shadow: rgba(0,0,0,.4) 0 1px 0;
    color: #ffffff;
    font-size: 24px;
```

```
    font-family: Georgia, serif;
    text-decoration: none;
    vertical-align: middle;
}
select {
  font-size: 32px;
}
input {
     border: 5px solid white;
     -webkit-box-shadow:
       inset 0 0 8px  rgba(0,0,0,0.1),
          0 0 16px rgba(0,0,0,0.1);
     -moz-box-shadow:
          inset 0 0 8px  rgba(0,0,0,0.1),
          0 0 16px rgba(0,0,0,0.1);
      box-shadow:
       inset 0 0 8px  rgba(0,0,0,0.1),
          0 0 16px rgba(0,0,0,0.1);
     padding: 15px;
     background: rgba(255,255,255,0.5);
      margin: 0 0 10px 0;
}
```

Der *cssid* -Befehl weist einer über die ID eines HTML-Elements ein entsprechendes CSS-Element zu. Die Zuweisung sollte unmittelbar nach der Erzeugung des HTML-Elements erfolgen. Beispielsyntax: *cssid htmlid(), "background-color: yellow;"*

Mit dem *cssclass* -Befehl werden alle Elemente einer Klasse (class) in den gewünschten CSS-Stil überführt. Die Referenz nennt als Beispiel die Anpassung aller Elemente der Klasse „Button" mit *cssclass "button", "background-color: yellow;"*

Im vorigen Beispiel erzeugt diese Zeile auf der Position vor *wait* einen gelben Button, wenn die erste Zeile auskommentiert ist, also der externe CSS-Stil entfällt.

2.8.6 CSS, HTMLID

Die ID eines HTML-Elements erlaubt den Zugriff von CSS und von JavaScript auf dieses Element. In Anlehnung an die vorigen Beispiele werden nun zwei „Button“-Elemente über diesen Weg im Aussehen geändert. Die Funktion *htmlid* liefert diese ID des zuletzt erzeugten Elements, der Aufruf direkt danach ist sinnvoll. Die CSS-Gestaltung gestaltet sich also wie folgt:

```
button “ROT”,[ok]
rt=htmlid()
button “GELB”,[ok]
ge=htmlid()
cssid rt, “background-color: red;”
cssid ge, “background-color: yellow;”
wait
[ok]
end
```

2.8.7 ONLOAD

Der mit *onload* festgelegte Programm-Abschnitt wird bei jedem Neuladen der Seite angesprungen.

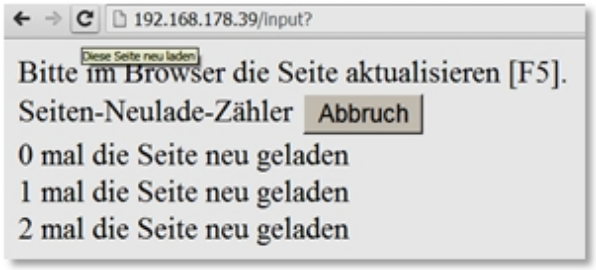


Abbildung 2-27: Test mit OnLoad

Damit lassen sich Benutzeraktivitäten auswerten, oder auf den HTML-Refresh reagieren. Im Beispiel wird gezählt, wie oft ein Windowsbenutzer die F5-Taste drückt, oder mit dem entsprechenden Icon oder Menüpunkt ein Refresh auslöst.

```
html “Bitte im Browser die Seite aktualisieren [F5].<br>”
html “Seiten-Neulade-Z&aumlhler “
button ” Abbruch”, [ok]
onload [refresh]
```

```
i = 0
wait
[refresh]
html “<br>” & i &” mal die Seite neu geladen”
i = i + 1
wait
[ok]
end
```

2.8.8 JavaScript und HtmlID, HtmlVar

ESPBASIC kann JavaScript generieren, welches dann vom Browser interpretiert wird. Ein einfaches Beispiel ist eine Warnung in JavaScript, ein sogenannter *alert* . Ein *alert* ist Teil des Browser-Fensters und kann mit *window.alert()* aufgerufen werden. Dies funktioniert auch von ESPBASIC aus, da die Ausgaben von *wprint* oder *html* im Browserfenster landen und selber per JavaScript HTML-Kode erzeugen. Ein kurzes Beispiel:

```
html |<script>|
html |alert(“Hier spricht JavaScript”);|
html |</script>|
```

Eingebettet in HTML-<*script*>- Tags übergibt die eine Scrip-Sprache (BASIC) an JavaScript und lässt den Ausdruck *alert(“Hier spricht JavaScript”);* interpretieren. Das Semikolon nach der Anweisung verrät, dass es sich hierbei nicht um BASIC handelt. Das Zusammenspiel wird interessant, wenn ein HTML5-Element ins Spiel kommt. Das *meter* -Element stammt aus HTML5 und steht in ESPBASIC zur Verfügung.

Sprachlich interessant ist auch, das ESPBASIC in C/C++ geschrieben ist, mit JavaScript selber eine Programmumgebung schafft und dem Benutzer all diese Möglichkeiten weiter reicht.

Es gibt viele Gründe im Browserfenster JavaScript zu verwenden, da diese Programme durch das Internet höchst effektiv mit dieser Script-Sprache umgehen. ESPBASIC selber generiert seine einfachen Grafiken ebenfalls mit Hilfe von JavaScript. Will man als Programmierer Werte von der

ESPBASIC-Welt in die JavaScript-Welt transportieren, ist eine Art Tunnel erforderlich. Mit den beiden Funktionen *htmlvar()* und *htmlid()* stehen diese Werkzeuge zur Verfügung.

Im nun folgenden kleinen Beispiel soll nur das Zusammenspiel dieser beiden Welten demonstriert werden. Im Abschnitt Anwendungen folgen noch aufwändigere Beispiele, die teilweise JavaScript aus *.js-Dateien einbinden und zeigen, dass JavaScript in sich schneller im Browser beim Anwender interpretiert wird, als das in ESP wohnende BASIC. Messung und grafische Darstellung im ms-Bereich sind dadurch im Browser möglich. ESPBASIC liefert nur die Messwerte mit seinen übersichtlich gestalteten Aufrufen zur Ein- und Ausgabe. Eine ESPBASIC-Variable bekommt den Wert 50 und wird mit dem *meter* -Element mit dem Wertebereich 0 bis 100 verknüpft, damit die Anzeige in der Mitte steht. Nach der Erzeugung bekommt die BASIC-Variable *id* den entsprechenden (von BASIC mittels JavaScript erzeugten HTML5-Element) Wert, der dann von BASIC aus wieder den eigenen JavaScript-Routinen zugeordnet werden muss, um den Wert des Elements zu erhalten. Ein HTML-Break bricht die Zeile nach dem *meter* -Element um und danach wird auf JavaScript umgeschaltet. Edgar Wallace nennt sich hier JavaScript und die Variable *name _element* erhält mit *getElementById* über *htmlvar()* die *id* vom *meter* -Element.

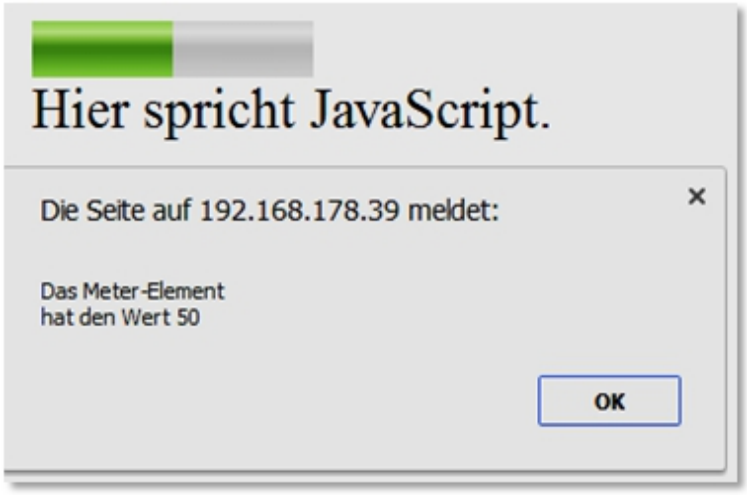


Abbildung 2-28: Austausch zwischen ESPBASIC und JavaScript

JavaScript unterscheidet bei den Funktionsnamen im Gegensatz zu ESPBASIC zwischen Groß- und

Kleinschreibung! Die Variable *wert* erhält den *value* des *meters*. Schließlich erscheint mit dem *alert* der Wert 50 mit JavaScript-Mitteln auf dem Schirm. Das Script wird geschlossen und bei Bedarf kann die *id* in BASIC ausgegeben werden. Wenn eine Seite nur einmal aufgebaut wird, ändert sich diese Variable nicht. Bei einem erneuten Aufbau kann sie sich jedoch unterscheiden.

```
x = 50
meter x,0,100
id = htmlid()
html |<br><script>document.write("Hier spricht JavaScript.");|
html |var name_element = document.getElementById("| & htmlvar(id) & |");|
html |var wert = name_element.value;|
html |alert("Das Meter-Element\nhat den Wert " + wert);|
html |</script>|
'print id
End
```

2.8.9 JAVASCRIPT UND JSCALL

Beide Funktionen sollen es ermöglichen mit JavaScript in *.js-Dateien einfach umzugehen. Bisher blieben die Versuche jedoch meist erfolglos. Die Zeilen

Javascript "jstest.js"

Jscall "jstest"

sollen eine Funktion *jstest* in der Datei *jstest.js* aufrufen, die wiederum nur ein *alert* darstellt:

function jstest()

{alert ("Dieses Script steht in einer Datei.");}

Ein Nachteil von JavaScript in ESPBASIC ist, dass das Script bei einem Fehler einfach aufhört, also ohne Meldung abbricht. Fehlersuche ist dadurch nicht ganz komfortabel. Die Datei ist im Dateisystem unter */uploads/jstest.js* abgespeichert. Parameterübergabe ist scheinbar nicht vorgesehen.

Da es einen alternativen Weg gibt, sollen dieser hier aufgezeigt werden. Bei *.js-Dateien aus dem Netz erfolgt die Einbindung auf einer HTML-Seite etwa so:

… src=“jstest.js” …

Liegt die Datei im ESP-Dateisysem unter /uploads/, dann ist die Syntax:

… src=”/file?file=jstest.js” …

Dann funktioniert auch der externe Aufruf einer JavaScript-Funktion:

html |<script src=”/file?file=jstest.js”|

html | type=“text/javascript”></script>|

html |<script type=“text/javascript”>|

html |jstest();|

html |</script>|

2.9 DEBUGGER, VARIABLEN

Ab einer BASIC-Installationsgröße von 2 MByte ist ein eingebauter Debugger vorhanden. In der allgemeinen Menüzeile erscheint dieser Punkt dann zwischen RUN und SETTINGS. Die kleineren Versionen für z.B. den ESP01 enthalten diesen Kode nicht. Mit zwei Anweisungen wird das Debug-Fenster vom Programm her unterstützt. Einmal kann mit der Anweisung *Debugbreak* der Programmlauf an einer bestimmten Stelle angehalten werden, die Fortsetzung erfolgt dann im Debug-Fenster mit CONTINUE. Mit der zweiten Anweisung *debug.log* ist es möglich, Text zu einem Log zu schicken. Das Fenster selber ist wie folgt aufgebaut:

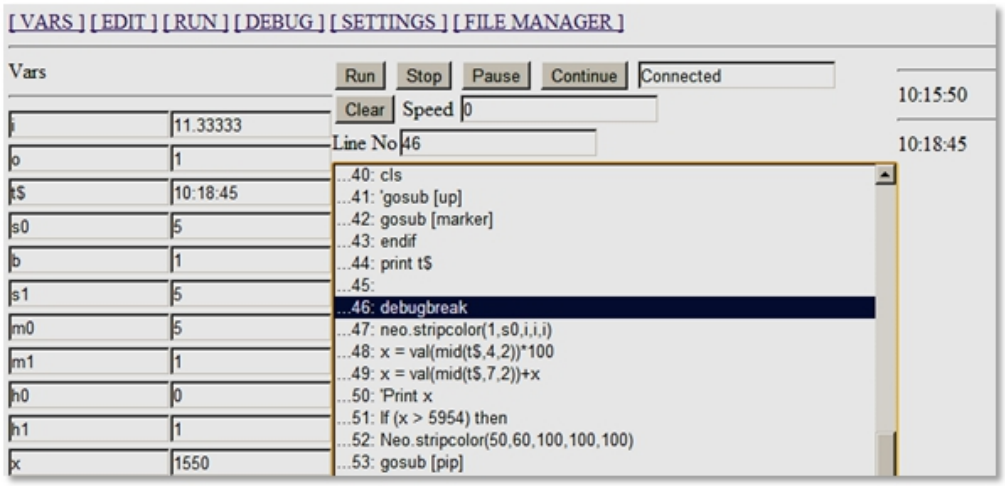


Abbildung 2-29: Der Debugger

Die Abbildung zeigt einen mit *DebugBreak* gesetzten Breakpoint bei der Ausführung eines Uhrenprogramms, wie es in den Anwendungen im hinteren Teil des Buches zu finden ist. Falls im Debugger eine Pause in Millisekunden (z. B. Speed 500) zwischen den Zeilen aktiviert ist, ist der ESP über die Browser-Schnittstelle nur noch sehr schwer erreichbar, wenn das Debug-Fenster nicht aktiv ist. Auch kann es vorkommen, dass in der Statusmeldung nach der Taste CONTINUE ein „Disconnected“ auf eine unterbrochene Verbindung hinweist. In einem solchen Fall kann gewartet werden, bis der ESP nach einem Absturz wieder hoch fährt und die Verbindung wieder aufgenommen werden kann. Falls

SETTINGS zugänglich bleibt, kann von dort ein RESET bei schwieriger Verbindung versucht werden, ansonsten ist der nächste Stromausfall abzuwarten. Als allerletzte Maßnahme wäre die physische Kontaktaufnahme mit dem Reset-Taster am ESP-Modul zu nennen. Damit sei belegt, dass der Debugger nur für wirklich unlösbar erscheinende Probleme herangezogen werden sollte. Abhilfe schafft in einem solchen Fall wirklich das Einfügen einer *debugbreak* -Anweisung. In einer Schleife oder Timer-Anweisung kann dann manuell mit CONTINUE ein weiterer Durchlauf initiiert werden, bis der Breakpoint wieder erreicht wird.

3 Spezielle Hardware/Bibliotheken

Eingebundene Softwarebibliotheken, wie Zeit und Software-Serial, sind im vorigen Abschnitt behandelt worden. Hardware-Bibliotheken, die zur Zeit der Kompilierung von ESPBASIC eingebunden wurden, sind Gegenstand der folgenden Ausführungen. Dabei sei erwähnt, dass ESPBASIC in verschiedenen Größen vorliegt und in den kleineren Versionen nicht alles unterstützt wird, wie ja auch die Grafikfunktionen nur ab 2 MByte verfügbar sind.

Im Wesentlichen handelt es sich um externe Elemente, wie Anzeigen mit LCD, OLED, TFT und Neopixel (WS RGB-LED), sowie weit verbreitete Temperatur- und Luftfeuchtesensoren. Mit den zwei seriellen Hardwareprotokollen I2C und SPI lassen sich mit wenigen Zeilen auch Hardwareerweiterungen ansprechen, die nicht fest eingebaut sind. Beispiele sind Frequenzgeneratoren, Analog-Digital-Wandler und mehr, wie im Teil Anwendungen zu lesen ist.

3.1 Temperatursensor DS18B20

Mit dem Dallas Semiconductor Temperatursensor DS18B20 kann mit dem Einzeiler

Print temp(0)

23.1875

Done…

in ESPBASIC der aktuelle Wert der Temperatur in °C ausgegeben werden. Der Parameter 0 ist dabei die Nummer des Sensors, da an derselben Leitung mehrere gleiche Sensoren angeschlossen sein können. In ESPBASIC ist als Anschluss Pin 2 fest vorgegeben.

Abbildung 3-1: Wasserdichte Version des Dallas Sensors DS18B20

Der DS18B20 ist sowohl als wasserdichte -, als auch als einfache Variante im TO-92 Gehäuse erhältlich. Die drei Anschlüsse sind von links nach rechts Gnd, Data, Vcc. Die wasserdichte Version wird mit verschiedenen Kabellängen geliefert, an deren Enden die Anschlüsse in den entsprechenden Farben Schwarz, Gelb und Rot gekennzeichnet sind. Die einbettende Metallhülse hat einen 6-mm-Durchmesser. Als Spannungsversorgung ist der Bereich 3,0 V bis 5,5 V angegeben, der Temperaturbereich mit -55 °C bis +125 °C bei einer Genauigkeit von 0,5 Grad zwischen -10 °C und 85 °C und der Auflösung 1/16 Grad (0,0625) in der ESPBASIC-Vorgabe. Eine Messung und Übertragung erfordert bis zu 750 ms. Bei Benutzung mehrerer Sensoren am selben Anschluss-Pin (2), kann laut Datenblatt ein Widerstand von 4,7k zwischen Vcc und Data erforderlich sein.

Mit *Textfeld* , *Meter* und *Timer* aus dem Abschnitt ESPBASIC Speziell dieses Buches, ist eine kontinuierliche Temperaturanzeige im Browserfenster schnell realisiert. Wegen der Kürze der Abfrage ohne irgendeine Initialisierung ist auch in jedem *msg* -Server noch Platz, so dass auch Messungen über HTTP direkt in z. B. Excel möglich sind.

3.2 OLED-ANZEIGE

Als eine Variante kommt hier eine kleine 1306-OLED-Anzeige zum Einsatz mit der Auflösung 128x64, die über zwei Leitungen mit dem ESP kommuniziert. Das Protokoll ist I2C, wodurch die Möglichkeit besteht mehrere Anzeigen über dieselben beiden Leitungen unterschiedlich anzusprechen.

Abbildung 3-2: OLED mit verschiedenen Schriftgrößen

Am kleinen ESP01 stehen nur die Leitungen 0 und 2 für I2C zur Verfügung, darum benutzen die Beispiele hier ebenfalls diese Anschlüsse für das Witty Cloud Modul. Für kleine ESPBASIC-Systeme gibt es aus Platzgründen laut Referenz nur drei rudimentäre Anweisungen *oledPrint* , *oledCls* und *oledSend* . Voreingestellt sind dabei die Anschlüsse 0 und 2. Der letzte Befehl ermöglicht eigene Kommandos des 1306-

Chips in einem kleineren ESPBASIC System zur Ausführung zu bringen.

3.2.1 OLED

Die größeren ESPBASIC-Binärdateien ab 2 MB beinhalten mehr vorinstallierte Anweisungen zum Umgang mit OLED-Anzeigen:

Farbe Schwarz/Weiß	*Oled.color()*
Bildschirm löschen	*Oled.cls()*
Linie	*Oled.line()*
Rechteck	*Oled.rect()*
Rechteck gefüllt	*Oled.rect.fill()*
Kreis	*Oled.circle()*
Zeichensatz	*Oled.font()*
Ausgabe	*Oled.print()*
Ohne Punkt an Pin 0/2	
Oled 0/2	*oledPrint*
Oled 0/2	*oledCls*
Oled 0/2	*oledSend*

Die Anweisung *i2c.setup(sdaPin,sclPin*) legt zu Beginn explizit die erwünschten Anschlüsse fest. Am Witty Cloud wäre z.B. auch Pin 5 und 4 zu benutzen. Das folgende Listing erzeugt eine „einmalige“ Grafik mit einigen Linien. Einmalig, da keine Wiederholungen programmiert sind.

```
i2c.setup(0,2)
oled.cls()
```

```
oled.line(0,32,128,32)
oled.line(64,0,64,63)
for x= - 32 to 32 step 0.1
 y = x * x/8
 oled.line(x+64,64-y,x+64,64-y)
next x
wait
```

Alle fünf Sekunden erscheint ein Kreismuster mittels *oled.circle():*

```
i2c.setup(0,2)‘sda,scl
timer 5000,[mach]
[mach]
oled.cls()
for r = 100 to 0 step -3
 oled.circle(64,32,r)
next r
oled.circle.fill(64,32,32)
wait
```

Das nächste Beispiel ist ein Zähler mit Textausgabe an Position *x = 1* und *y = 1* mit Änderung der Zeichensatzgröße und Doppelausgabe in Schwarz und Weiß, um die Lesbarkeit zu wahren. Der Texthintergrund kann nicht mit eingebauten Aufrufen gesetzt werden.

```
i2c.setup(0,2)
oled.cls()
oled.font(16)
timer 100,[mach]
i = 0
wait
[mach]
oled.color(0)
oled.print(str(i),1,1)
oled.color(1)
i = i + 1
oled.print(str(i),1,1)
wait
```

Die Positionierung von Text unter Verwendung der drei unterschiedlichen Schriftgrößen zeigt ein Beispiel zur Zeitdarstellung auf der OLED-Anzeige.

Es wird die interne *Time* -Routine benutzt, die nur bei Internetzugang über z.B. einen Router die Zeit liefert. Im hinteren Teil des Buches wird eine andere Zeitanzeige, eine Dezimaluhr mit *NeoPixel* -Streifen, dargestellt. Die drei Schriftgrößen sind in einem Array gespeichert, um sie nacheinander mit der Variablen *i* abrufen zu können und sie bei der Positionierung entsprechend zu berücksichtigen. Die Übersetzung des Wochentags (dow - day of week) erfolgt per einfacher Verzweigung.

```
dim s(3)
s(1) = 10
s(2) = 16
s(3) = 24
i = 1
i2c.setup(0,2)
timer 5000,[mach]
[mach]
oled.cls()
oled.line(28,0,28,63)
oled.line(0,38,127,38)
oled.font(s(i))
z$ = time("hour:min:sec")
if i < 3 then
 t$=time("dow")
 if t$="Mon" then d$="Montag"
 if t$="Tue" then d$="Dienstag"
 if t$="Wed" then d$="Mittwoch"
 if t$="Thu" then d$="Donnerstag"
 if t$="Fri" then d$="Freitag"
 if t$="Sat" then d$="Sonnabend"
 if t$="Sun" then d$="Sonntach"
else
 d$ = time("day")
endif
```

```
oled.print(d$,32,32-s(i))
oled.print(z$,32,40)
i = i + 1
if i > 3 then i = 1
wait
```

Abbildung 3-3: OLED-Grafik mit LDR- und Taster-Abfrage

Noch ein etwas längeres Beispiel - schon fast eine Anwendung - zum Schluss des Abschnitts über OLED-Anzeigen. Es benutzt das Board Witty Cloud mit dem dort vorhandenen LDR und Taster. Die Taster Abfrage benutzt den Interrupt und löst eine Helligkeitsmessung aus, die die Ergebnisse graphisch auf dem Display zur Anzeige bringt und so lange anzeigt bis ein erneuter Tastendruck den Vorgang wiederholt. Der überwiegende Teil des Listings besteht aus der Darstellung des Diagrammrasters. Da die Linienfunktion einzelne Pixel setzen soll, ist keine weitere Verzögerung mehr notwendig, um den Aufbau zu beobachten. Danach startet die Messung und die einzelnen Messwerte oder Messpunkte werden per Linie miteinander verbunden.

```
interrupt 4,[taster]
gosub [messen]
Wait
[messen]
oled.cls()
for x = 16 to 112 step 16
For y = 0 to 63 step 2
Oled.line(x,y,x,y)
Next y
Next x
```

```
for y = 16 to 56 step 16
For x = 0 to 127 step 2
Oled.line(x,y,x,y)
Next x
Next y
Oled.rect(0,0,127,63)
For x = 0 to 127 step 1
y = 64-io(ai)/1024*64
if x>0 then oled.line(x-1,oy,x,y)
delay 100
oy=y
Next x
oled.print("Any Key",1,50)
return
[taster]
gosub [messen]
Wait
```

3.2.2 OLED FÜR ESP01

Mit drei weiteren Befehlen kann sofort und ohne Initialisierung ein 1306-Display an den voreingestellten Pins 0 und 2 (sda, scl) zur Ausgabe genutzt werden. Mit *oledprint* – ohne Punkt – Erfolgt eine schnelle Ausgabe von Zeichenketten optional mit einer Positionsangabe für x und y.

Die Einbindung dieser drei Kommandos soll die Verwendung von OLED-Displays auch auf kleineren ESPBASIC-Systemen ermöglichen. Das Listing beschreibt das komplette 128x64 Display im Zeichensatz 8x8, so dass 8 Zeilen mit 16 Zeichen erscheinen. In der ersten Zeile steht die Uhrzeit zweimal hintereinander. Der Timer wiederholt die Gesamtausgabe.

```
timer 1000,[tm]
[tm]
a$=time("hour:min:sec") & time("hour:min:sec") &
"1234567890123456"&"2345678901234467"&"3456789012345678"&"4567890
123456789"&"5678901234567890"&"6789012345678901"&"789012345678901
2"
oledprint a$
wait
```

Oledcls ist zum Löschen des Bildschirms vorgesehen. Ein Testprogramm für beide Anweisungen und Positionsangaben zeigt eigensinniges Verhalten, was vermutlich Exemplarabhängig ist.

```
oledcls
for x = 1 to 40 step 8
 oledprint “HALLO”, x, x
 delay 500
next x
```

Mit *oledsend* kann der Programmierer Kommandos direkt zum Display senden. Das Datenblatt zum 1306-Chip gibt diese Kommandos in hexadezimaler Schreibweise an. Ein einfaches Kommando ist die Invertierung des Bildschirms. Mit dem Einbyte-Kommando *A6* bzw. *A7* wird zwischen den beiden Darstellungen umgeschaltet. Als kleiner Test folgt ein OLED-BLINK in 7 Zeilen:

```
timer 500,[invert]
i = 0
wait
[invert]
oledsend hextoint(“a7”)+i
i = not i
wait
```

Die Variable *i* wechselt zwischen *0* und *-1* .

3.3 LCD DISPLAY

Für ein 1602-LDC-Display mit I2C-Ansteuerung, stellt ESPBASIC vier Aufrufe zur Verfügung:

lcdPrint

lcdCls

lcdBl

lcdSend

Die stark verbreiteten, zweizeiligen LCD-Displays vom Typ 1602 mit jeweils 16 Zeichen pro Zeile sind inzwischen als I2C-Varianten zu erhalten. Auch gibt es sehr preiswerte Adapterplatinen, die ein solches LCD1602 mit seinen 16 Anschlüssen auf Zweidraht-I2C umwandeln. Zusätzlich erfolgt dort die Einstellung des Kontrasts mittels Potentiometer sehr komfortabel. Die Print-Anweisung gibt eine Zeichenkette an der Position x/y aus auf dem Display, wobei der Koordinatenursprung, je nach Typ, unterschiedlich sein kann.

Abbildung 3-4: LCD Anzeige mit I2C-Interface

Ein Test wäre ein *lcdprint „12345“, 1, 1* , um zu sehen, wo die Ausgabe im konkreten Fall erfolgt. Mit *lcdcls* löscht ESPBASIC die Anzeige und *lcdbl 1* schaltet die Hintergrundbeleuchtung (BackLight) an oder mit *0* aus. *Lcdsend* dient dazu direkt mit dem Display zu kommunizieren. Dazu sieht der Befehl einen Wert und einen von drei Modi als Parameter vor. Die drei Modi sind 0 = Command, 1 = Data, 2 = FOUR_BITS. Die Syntax lautet *lcdsend wert, modus.* Mit dem Einzeiler

Lcdprint time(),0,0

erscheint auf dem Display Datum und Uhrzeit, wie sie die Funktion *time* () ohne Parameter liefert. Die I2C-Adresse lautet 39 und ist fest in ESPBASIC kodiert. Das folgende Beispiel zeigt, wie mit *lcdsend()* horizontales Scrolling mit

dem Kommando „1C“ aus dem Datenblatt des 1602 erfolgt und damit einen weiteren Befehl hinzufügt.

```
‘adr 0x27 (39) default
‘i2c.setup(0,2)
Lcdbl 1
timer 1000,[messen]
wait
[messen]
lcdprint time(“hour:min:sec”),0,0
lcdprint time(“hour:min:sec”),16,0
lcdsend hextoint(“1C”),0 ‘>>>>>
wait
```

Abbildung 3-5: Wetter und Zeit in wenigen Zeilen

Mit den Ausführungen aus Abschnitt 2 und der *OpenWeatherMap* -Abfrage ist eine kleine Wetterstation möglich. Dabei fällt das Listing etwas länger aus, weil persönliche Vorlieben berücksichtigt sind:

- Ort im Browser zur Laufzeit änderbar
- Windrichtung nicht in Grad sondern als Süd-West 5 (SW5)
- Windgeschwindigkeit in Beaufort
- Aktualisierung alle 15 * 63418 Millisekunden
- Blinkender Sekundendoppelpunkt

Neben dem Ort lassen sich auch alle anderen Variablen mit Textfeldern versehen, um den Überblick auch im Browser zu sehen. Hier würde das Listing jedoch unnötig lang.

```
x = 0
ort = “unterbach,de”
timesetup(1)
Textbox ort
button “Ok”,[ccb]
```

```
Lcdbl 1
Lcdcls
timer 1000,[tm]
Timercb 15 * 63418,[cb]
Gosub [cb]
wait
[ccb]
Gosub [cb]
wait
[tm]
lcdprint time(“hour:min”),11,1
x = not x
lcdprint chr(58 + (26 * x)),13,1
wait
[cb]
url = “api.openweathermap.org/”
msg = “data/2.5/weather?q=”
api = “&APPID=5960f9a3ccf9b92…….”
a = wget(url&msg&ort&api)
temp = val(json(a,“main.temp”))-273.00
tmp = str(int(temp*100)/100)&“C”
city = json(a,“message.name”)
prs = json(a,“main.pressure”)
spd = json(a,“main.speed”)
deg = json(a,“main.deg”)
B = floor(pow((spd/0.836),0.66))
lcdcls
lcdprint city,0,1
dir = round(deg/45)
w = “N NOO SOS SWW NW”
W = mid(w,1+(dir * 2),2)
LcdPrint (prs)&” “&W&B&” “&spd&” “&tmp,0,0
return
```

Dies alles funktioniert nur, wenn der ESP8266 Zugang zum Internet hat.

3.4 NeoPixel - WS2812 RGB-LED

Leuchtdioden mit drei Farben und eigenem kleinen Controller werden NeoPixel genannt und sind z. B. vom Typ WS2812. Sie sind üblicherweise hintereinander in Kreisen oder als Streifen angeordnet. Alle hier angeführten Untersuchungen der ESPBASIC Funktionen dieser Hardware benutzen einen Streifen mit 60 LED mit einer Länge von einem Meter.

Abbildung 3-6: Neopixel alias WS2812

Im Teil Anwendungen kommt dieser Streifen bei der Dezimaluhr des Rheinturms (Düsseldorf) zum Einsatz, da dort zur Darstellung der Zeit mindestens 39 Leuchten notwendig sind. Die NeoPixel lassen sich aus ESPBASIC heraus mit folgenden Anweisungen benutzen:

Neo.setup()

Neo()

Neo.cls()

Neo.stripcolor()

Neo.shift()

Neo.hex()

NeoPixel benötigen genau einen Pin, der die Daten empfängt und an die folgenden Pixel weiter reicht. Mit *neo.setup(0)* wird Pin 0 zur Datenleitung. Zu Beginn kann es nötig sein mit *neo.cls()* alle Pixel zu löschen bzw. allen die Helligkeit 0 zuzuweisen. Mit der Basisanweisung *neo* kann dann jedes Pixel einzeln einen RGB-Farbwert erhalten. Ein letzter Parameter legt fest, ob die Ausgabe nur vorgespeichert wird (1) und erst noch andere Pixel gesetzt werden oder sofort erfolgt. Mit *neo(3,127,127,127,1)* bereitet Pixel *3* vor auf halbe Helligkeit mit gleichen Rot-, Grün- und Blau-Anteilen, als mittleres Weiß. Eine anschließende Anweisung *neo(10,127,0,0,0)* setzt die 10. LED auf „Halbrot“ und sorgt mit dem letzten Parameter „0“ dafür, dass alle anderen vorbereiteten Ausgaben erfolgen. Sollen nebeneinander liegende Pixelbereiche gleich erscheinen, bietet sich *neo.stripcolor()* an. Als Parameter werden Anfangs-Pixel und End-Pixel, sowie die Farbanteile R, G, und B jeweils im Bereich 0 bis 255 übergeben. Der Aufruf *neo.stripcolor(10,19,0,0,255)* schaltet Pixel 10 bis 19 auf das intensivste Blau. Spielereien optischer Art erlaubt *neo.shift()*. Damit lassen sich bis zu 20 hintereinander liegende Pixel jeweils einmal hin- oder her schieben. Mit Wiederholungen lassen sich so interessante Effekte erreichen. Als Beispiel schiebt *neo.shift(10,15,1)* den angegebenen Pixel-Bereich um eine Position in die eine Richtung, bei Schluss Parameter *-1* erfolgt die Schiebung in entgegengesetzter Richtung. Mit *neo.hex()* liegt ein weiterer spezieller Befehl vor, dessen Anwendung sich aus der Referenz nicht sofort erschließt.

Zu Beginn steht ein Test der ersten drei Pixel mit voller Helligkeit und Ampel-Illumination. Dabei legt die erste 0 die Datenleitung fest und die letzte 0 sorgt dafür, dass alle Pixel gleichzeitig angeschaltet werden. Anhand der Helligkeit lässt sich abschätzen, dass die Spannungsversorgung der Pixel nicht vom ESP erfolgen sollte. Würden aus Versehen alle 60 Pixel mit voller Helligkeit (255) leuchten und ein Strom von 20 mA pro LED-Farbe veranschlagt, berechnet sich der Strombedarf zu 60x20x3 mA, also 3,6 Ampere. An 5 Volt ergibt das 18 Watt!

neo.setup(0)

```
neo.cls()
neo(0,255,000,000,1)
neo(1,255,255,000,1)
neo(2,000,255,000,0)
```

Ein anderes Beispiel testet die 60 Pixel an Pin 0 nur sehr kurz mit voller Helligkeit auf eigenes Risiko. Ohne Verzögerung entsteht ein kurzer heller Blitz. Sollten bei *neo.cls()* aus Versehen die Klammern vergessen werden, stoppt das Programm mit einer Fehlermeldung, die LED leuchten aber mit 18 Watt weiter…

```
neo.setup(0)
neo.stripcolor(0,59,255,255,255)
neo.cls()
```

Ein Schiebetest als Unterprogramm für den gesamten Streifen in 20er Häppchen:

```
[schieber]
for z = 1 to 60
neo.shift(41,60,-1)
neo.shift(21,40,-1)
neo.shift(1,20,-1)
next z
return
```

Alles zusammen gepackt und noch etwas angereichert ergibt sich eine bunte Animation, die sich endlos alle fünf Sekunden wiederholt. Nach dem kurzen hellen Blitz folgt ein blaues Dimmen, wiederum gefolgt von einer zufälligen Farb- und Pixelzuordnung, allerdings ohne direkte Anzeige. Erst mit *neo(0,0,0,0)* erfolgt die zufällige Darstellung. Das Unterprogramm schiebt dann alle 60 Pixel nach unten, wenn der Streifen so senkrecht angeordnet ist, dass Pixel 0 unten liegt.

```
neo.setup(0)
timer 5000, [tm]
wait
[tm]
neo.stripcolor(0,59,255,255,255)
```

```
for i = 64 to 2 step -2
neo.stripcolor(0,59,0,0,i)
next i
for i = 1 to 100
 neo(rnd(60),rnd(64),rnd(64),rnd(64),1)
next i
neo(0,0,0,0,0)
gosub [schieber]
wait
[schieber]
for z = 1 to 60
neo.shift(41,60,-1)
neo.shift(21,40,-1)
neo.shift(1,20,-1)
next z
return
```

3.5 TFT DISPLAY

Das TFT-Display mit Touch-Oberfläche mit dem ILI9341-Chipsatz wird nativ von ESPBASIC mit außergewöhnlich vielen Kommandos und Funktionen unterstützt. Der Autor von ESPBASIC hat in der englischen Referenz auf über zehn Seiten dieses Display behandelt. Von der Beschaltung bis hin zu eigens konstruierten Elementen für eine eventuelle Benutzeroberfläche auf diesem Display ist dort alles ausführlich und übersichtlich dargestellt. Um den Rahmen hier zu begrenzen erscheint an dieser Stelle darum nur ein Überblick der Funktionen. Details dazu findet der Besitzer eines solchen Displays online im Original unter:

https://www.esp8266basic.com/language-reference.html

Das mit acht Leitungen beschaltete Display mit einer Auflösung von 320 x 240 Pixeln benötigt die ESP-Anschlüsse GPIO 4 (D/C), 12 (MISO), 13 (MOSI), 14 (SCK) und 16 (CS), sowie Masse und 3,3 Volt (VCC/RESET/LED) um per SPI-Protokoll mit dem ESP zu kommunizieren. Vorsicht: Das Witty Cloud Board führt am Pin Vcc nicht 3,3 – sondern 5 Volt!

TFT SETUP

tft.setup(CSpin,Dcpin,rotation)

TFT GRAFIK

tft.rgb(rt, gn, bl)

tft.fill(farbe)

tft.cls()

tft.line(x1 ,y1, x2, y2, farbe)

tft.rect(x, y, w, h, farbe)

tft.rect.fill(x, y, w, h, farbe)

tft.rect.round(x, y, w, h, radius, farbe)

tft.rect.round.fill(x, y, w, h, radius, farbe)

tft.circle(x, y, radius, farbe)

tft.circle.fill(x, y, radius, farbe)

tft.text.color(farbe)

tft.text.cursor(x, y)

tft.text.size(1 – 8)

tft.print(text)

tft.println(text)

tft.demo()

tft.text.font(typ 0 – 4)

tft.bmp(dateiname, x, y, hintergrund)

TFT GUI – Objekte der Benutzeroberfläche

tft.obj.button(text, x, y, w, h, textsize, forecolor, backcolor)

tft.obj.label (text, x, y, w, h, textsize, forecolor, backcolor)

tft.obj.checkbox(text, x, y, w, h, textsize, forecolor, backcolor)

tft.obj.radio(text, x, y, h, checked, textsize, forecolor,

backcolor)

tft.obj.toggle(icon1, icon2, x, y, checked, scale, backcolor)

tft.obj.bar(text, x, y, w, h, textsize, forecoloer, backcolor)

TFT GUI – Funktionen der Objekte zur Laufzeit

tft.obj.setlabel(id, text)

tft.obj.setvalue(id, wert)

tft.obj.setchecked(id, 1 oder 0)

tft.obj.invert(id)

tft.obj.getlabel(id)

tft.obj.getvalue(id)

tft.obj.getchecked(id)

Einige TFT-Displays mit XPT2045-Chip unterstützt ESPBASIC mit entsprechenden Funktionen. Die Kommunikation erfolgt über dieselben SPI-Leitungen. Der zusätzliche T_CS-Anschluss am ESP ist per Software frei wählbar und könnte GPIO 15 sein. Die zusätzlichen Anschlüsse T_CLK/SCK, T_CS/GPIO15, T_DIN/MOSI, T_DOUT/MISO sind in der Referenz abgebildet.

TFT GUI – Touch-Interface

tft.touch.setup(T_CS pin)

tft.touchx()

tft.touchy()

tft.checktouch()

---

touchbranch [sprungmarke]

Das in der Referenz angegebene komplette Beispiel wird hier nur aus Gründen der Vollständigkeit unverändert dargestellt:

```
tft.setup(16, 4, 3)
tft.touch.setup(15)
but1 = tft.obj.button(“PUSH”, 5,5,120,50,3)
chk1 = tft.obj.checkbox(“Check Me”, 5,100,40,1, 3)
lab1 = tft.obj.label(“press”, 0,190,200,24,3)
touchbranch [touchme]
wait
[touchme]
touch_obj = tft.checktouch()
serialprintln “checktouch ” & touch_obj
if touch_obj = but1 then
  tft.obj.invert(but1)
  tft.obj.setlabel(lab1, “button”)
endif
if touch_obj = chk1 then
  tft.obj.setlabel(lab1, “checkbox”)
  tft.obj.invert(chk1)
endif
return
```

3.6 IR – Infrarot Sender/Empfänger

Durch die Unterstützung von Infrarot-Sendern und Empfängern entsteht die Möglichkeit mit sehr preiswerten Mitteln Dinge zu Steuern, die auf solche Signale reagieren. Die eingebundene IR-Bibliothek sendet und empfängt dabei in einem weit verbreiteten digitalen Format und erkennt dadurch die Kodes von NEC und SONY.

Abbildung 3-7: IR-Sender und Empfäner-Diode

Somit ist es zum Beispiel möglich das Signal einer Fernbedienung zu empfangen und zu speichern, um mit dem eigenen IR-Sender die Fernbedienung zu ersetzen. Auch eigene Daten lassen sich über diesen Weg transportieren. Die dekodierten Empfangsdaten sind hexadezimale Zeichenketten in dem Format „4907d8e5:NEC:32“, wobei der erste Teil der eigentliche Inhalt ist und der Rest der Antwort aus dem Kode und der Anzahl der Bits besteht. Die IR-Routinen in ESPBASIC sind:

Ir.recv.setup()

Ir.recv.get()

Ir.recv.full()

Ir.recv.dump()

Irbranch

Ir.send.setup()
Ir.send.nec()
Ir.send.sony()

Sender und Empfänger benötigen jeweils einen Daten-Pin, der sich mit der jeweiligen Setup-Routine festlegen lässt. Die Empfangsabfrage gestaltet sich sehr komfortabel über einen *irbranch* , also über einen ereignisgesteuerten Programmteil, auf ESPBASIC-Art. Während *ir.recvget()* nur die Daten einer Übertragung liefert, gibt *ir.recv.full()* zusätzlich den Kode und die Anzahl der empfangenen Bits zurück. Die *ir.recv.dump()* - Funktion gibt Informationen über den seriellen Ausgang (TX) aus.

Abbildung 3-8: Einige Infrarot Fernbedienungen

Zur Überprüfung der Routinen sollen zunächst einige Fernbedienungen zum Einsatz kommen. Dazu wird der dreibeinige Sender (DATA/GND/VCC in der Vorderansicht) am Witty Cloud so angeschlossen, dass die Spannungsversorgung und die Masse an den Anschlüssen abgegriffen wird. Der Sender soll mit 3,3 – 5 Volt arbeiten. Als Daten-Pin wird GPIO05 benutzt.

Ein erstes Testprogramm versucht die Empfangsdaten im Browser anzuzeigen.

```
html “IR Test 1 <br>”
ir.recv.setup(5)
irbranch [ir]
wait
[ir]
a = ir.recv.get()
b = ir.recv.full()
print a
print b
return
```

IR Test 1

---

f7a05f

---

f7a05f:NEC:32

---

ffffffff

---

ffffffff:NEC:0

Eine dieser kleinen IR-Fernbedienungen liefert demnach einen NEC-Kode mit den Daten „f7a05f“, also drei Bytes in hexadezimaler Schreibweise. Scheinbar wird ein weiteres Byte bei der Übertragung benutzt, da ja 32 Bits angegeben werden. Die Daten mit 0 Bits sind vermutlich undefinierte Daten und könnten programmtechnisch abgefangen werden.

Im zweiten Test soll die LED an Pin 2 umschalten, wenn dieser Kode empfangen wird. Dazu wird eine Verzweigung eingebaut und die Ausgabe-Routine entsprechend aufgerufen. Die Negation *not* sorgt für das Umschalten. Bei der vorliegenden „Remote-Control“ handelt es sich um die Taste mit der Aufschrift „Mode“.

```
html “IR Umschalter:<br>MODE auf der FB<hr>”
ir.recv.setup(5)
irbranch [ir]
x = 0
wait
```

```
[ir]
if ir.recv.get() = “f7a05f” then
 io(po,2,x)
 x = not x
endif
return
```

Als Sender dient eine gewöhnliche Infrarot-Diode (IR-LED) mit Vorwiderstand, wie bei einer normalen LED. Bei 3,3 Volt sind 100 Ohm empfohlen, um eine gewisse unsichtbare Helligkeit zu erreichen und damit die gewünschte Entfernung überbrückbar zu machen. Für Testzwecke reichen auch höhere Werte, wie der hier gerade verfügbare 470 Ohm Widerstand. Die Setup-Routine legt GPIO04(D2), gleich neben GPIO05(D1) als Sende-Pin fest. Die erste IR Sendung soll das laufende Gerät mit dem entsprechenden Datenpaket der Power-Taste ausschalten.

Für das echte Power-Button-Feeling soll ein solches HTML-Element im Browser-Fenster bei Betätigung den entsprechenden Datensatz „f77887“ der vorliegenden Fernbedienung senden. Nach dem Start und entsprechender Ausrichtung der Sendediode, schaltet das Gerät mit Hilfe dieser 6 Zeilen ESPBASIC-Programm aus und an.

```
button ” POWER “,[power]
ir.send.setup(4)
wait
[power]
ir.send.nec(“f77887”,32)
wait
```

Eine Fernbedienung von Sony liefert als Sender Daten in diesem Format:

6C9B:SONY:15

549D:SONY:15

5BC:SONY:12

EBC:SONY:12

3.6.1 DATENÜBERTRAGUNG MIT LICHT

Als kleine Anwendung zu den Infrarot-Befehlen in ESPBASIC soll die aktuelle Uhrzeit auf diesem Weg gesendet und wieder empfangen werden. Das kann mit Hilfe von zwei getrennten ESP-Moduln geschehen, muss aber nicht. Der Sender kann zur Kontrolle seine eigene Ausstrahlung wieder per IR-Empfänger lesen. Die Ausstrahlung erreicht trotzdem alle Geräte in der Ausleuchtungszone der IR-LED als Sender. Wie in den Ausführen zu IR weiter oben bereits mit sehr kurzen Beispielen überprüft, ist lediglich jeweils ein Pin für Sender und Empfänger notwendig. Auch an dieser Stelle finden die beiden Anschlüsse GPIO05(D1) und GPIO04(D2) Anwendung. Hinzu kommen die Testroutinen zu den Zeitfunktionen aus dem entsprechenden Abschnitt weiter vorne. Als Kombination entsteht ein Programm, welches die aktuellen Werte aus der intern per Internet synchronisierten Zeit erfragt und in ein Format wandelt, welches dem NEC-Kode entspricht. Die Uhrzeit, um z. B. eine Sekunde vor Mitternacht, soll zur Übertragung mit 32 Bit in die Form „235959“ gebracht werden, so dass diese Zeichenkette als Hexadezimaldaten von der Senderoutine übernommen werden können.

Das Programm besteht aus einer Initialisierung mit Überschrift und zwei ESPBASIC „Branches“ (ereignisgesteuerte Sprungmarken). Der eine Branch ist die Timer-Routine, mit einem 1000 ms-Intervall, so dass pro Sekunde einmal die Zeit erfragt, gewandelt und gesendet wird. Die zweite Sprungmarke ist die des *irbranch* , die angesprungen wird, wenn IR-Empfangsdaten vorliegen. Die Anzeige im Browser erfolgt nur, wenn das Datenpacket mehr als ein Bit umfasst. Damit nur immer 10 Empfangs-Datensätze im Fenster erscheinen, sorgt ein Zeilenzähler *Zeile* dafür, dass beim Erreichen von *Zeilen* der Bildschirm mit *cls* gelöscht wird. Aus ästhetischen Gründen erfolgt die Darstellung der hexadezimalen Empfangsdaten in Großbuchstaben, was bei Ziffern wie in diesem Fall, keinen Unterschied macht.

ir.recv.setup(5)

ir.send.setup(4)

irbranch [rx]

html | <font family = “Helvetica” size = “-2”>|

```
html “Aktuelle Uhrzeit als Datenpaket. <br>”
timer 1000,[tm]
Zeile = 1
Zeilen = 10
wait
[rx]
if Zeile % Zeilen = 0 then cls
b = ir.recv.full()
bits = val(mid(b,instrrev(b,”:”)+1))
if bits > 0 then
 print upper(b)
 Zeile = Zeile + 1
endif
return
[tm]
t = time(“hour”)&time(“min”)&time(“sec”)
ir.send.nec(t,32)
wait
```

3.7 DHT – TEMPERATUR-/FEUCHTE-SENSOR

DHT11 und DHT22 sind zwei sehr verbreitete Sensoren für Temperatur und Luftfeuchte. ESPBASIC unterstützt diese Sensoren mit eigenen vier Funktionen oder Befehlen, die auf der DHT-Bibliothek beruhen. Beide Sensoren liefern Messwerte über nur einen Anschluss, wobei DHT11 nur ganzzahlige Werte -, der DHT22 auch Messwerte mit Nachkommastellen, bereitstellt.

Initialisierung	*dht.setup()*
Temperatur	*dht.temp()*
Luftfeuchte	*dht.hum()*
Gefühlte Temperatur	*dht.heatindex()*

Mit *dht.setup(model,pin)* initialisiert sich die Bibliothek und liefert die Daten entsprechend dem angegebenen Modell am angegebenen Pin. Die Temperatur in °C erfragt man mit *dht.temp()* und die relative Luftfeuchte (Humidity) in % liefert der Aufruf *dht.hum()* .

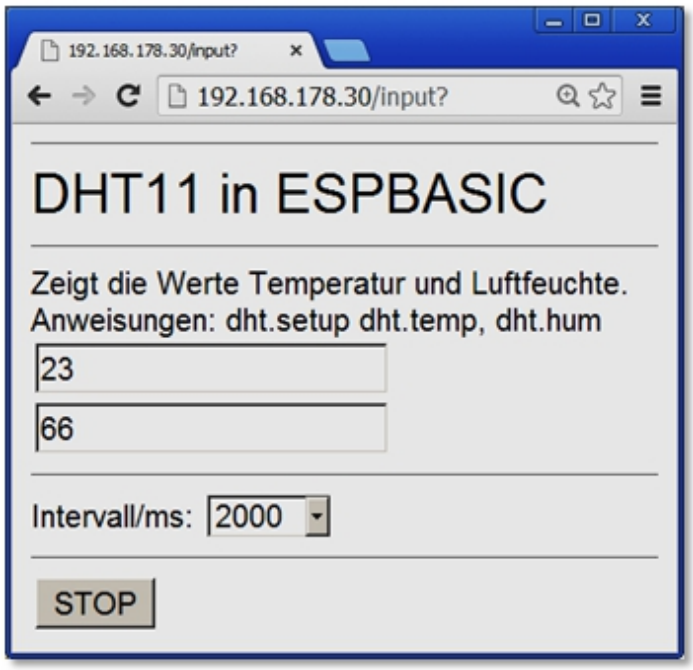


Abbildung 3-9: Unterstützter Sensor der DHT-Reihe

Die Bibliothek errechnet aus diesen Daten den Wert der gefühlten Temperatur, wie er auch in Wetterberichten noch auftaucht mit *dht.heatindex()* . Mit wenigen Zeilen lässt sich in ESPBASIC die unten dargestellte Benutzeroberfläche mit Anzeige und Messintervall Wahl erstellen. Dabei ist die gewünschte serifenlose Schrift auf dem Bildschirm zu großen Teilen am Umfang des Listings beteiligt.

Das folgende Listing erzeugt diese Ausgabe des Sensors DHT11 an Pin 2.

```
dht.setup(11, 2)
wprint “<hr noshade size=1><font size=5 face=Helvetica>”
wprint “DHT11 in ESPBASIC<br>”
wprint “<font size=2>”
wprint “Zeigt die Werte Temperatur und Luftfeuchte.”
wprint “<br>Anweisungen: dht.setup dht.temp, dht.hum<br>”
textbox t
textbox h
```

```
wprint “<hr noshade size=1>”
bla = 2000
wprint “Intervall/ms:   ”
dropdown bla,“2000,5000,10000,20000,30000,60000”
wprint “<hr noshade size=1>”
button “STOP”,[ende]
wprint “</font>”
timer bla,[messen]
wait
[messen]
t = DHT.TEMP()
h = DHT.HUM()
‘i = DHT.HEATINDEX()
‘t = 22 + rnd(10) - 5
‘h = 50 + rnd(50) -25
timer bla,[messen]
wait
[ende]
wprint “Messung beendet.”
end
```

Sollte gerade kein Sensor verfügbar sein, kann in der Messroutine durch Entfernung der Hochkommas auf Zufallswerte zurückgegriffen werden.

3.8 I2C VERBINDUNG

„Der I2C-Bus (Inter-IC-Bus) ist ein Bussystem bestehend aus zwei Leitungen zur Verbindung mehrerer ICs untereinander. Er wird meist zum Datenaustausch zwischen ICs innerhalb eines Geräts benutzt. Der besondere Vorteil liegt in der Verwendung von nur zwei Leitungen, einer Datenleitung SDA und einer Taktleitung SCL. Der Bus arbeitet prinzipiell nach dem Schieberegister-Prinzip, wobei Daten auf einer Datenleitung durch Taktimpulse in einen Empfängerbaustein geschoben werden. Unterschiedliche Bausteine werden über Adressen angewählt, die ebenfalls über den Bus gesendet werden.

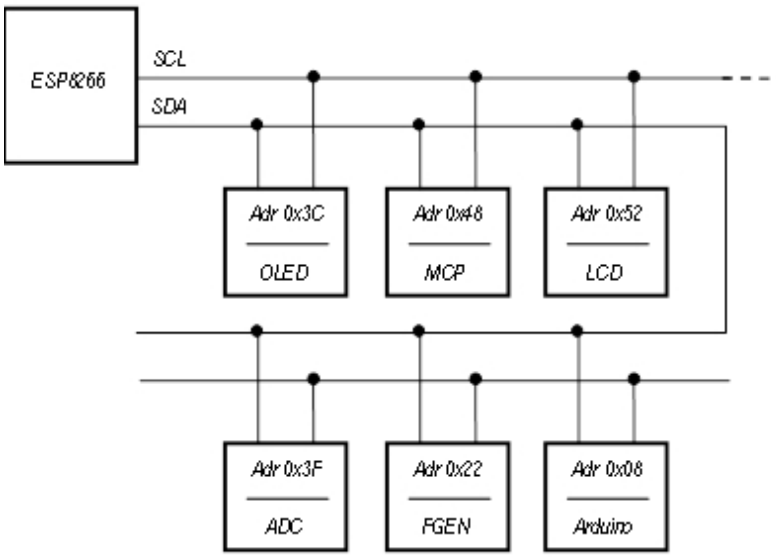


Abbildung 3-10: Teilnehmer am I2C-Bus

Es gibt zahlreiche Bausteine wie Digitalports, AD/DA-Wandler, Anzeigentreiber, Uhrenbausteine und Speicher, die das I2C-Busprotokoll beherrschen. Sie lassen sich vorteilhaft für Hardware-Erweiterungen einsetzen. Insbesondere kann man mit einem sehr einfachen Kabel viele Bausteine verketten. Mit Masse und Betriebsspannung benötigt man nur vier Leitungen.“ Soweit das Zitat aus [4].

ESPBASIC erlaubt es mittels des integrierten Befehlssatzes direkt Geräte an diesen zwei I2C-Leitungen anzusprechen.

Durch eine Adresse im Bereich von 1 bis 127 lässt sich jeder Busteilnehmer einzeln über dieselben Leitungen auswählen. Auch das hier benutzte OLED-Display ist ein Teilnehmer am I2C-Bus. Eine entsprechende Erweiterung der Hardwaremöglichkeiten erfolgt als Beispiel im Teil Anwendungen mit einem 16-Bit-A/D-Wandler und einem 12-Bit-D/A-Wandler. Es wäre dort also möglich die OLED-Anzeige ohne Zusatzanschlüsse seitens ESP in das Projekt einzubinden, weil beide auf unterschiedliche Adressen reagieren. Hier die Auflistung und Kurzbeschreibung der Befehle in ESPBASIC:

i2c.setup(sda,scl)

---

i2c.begin()

---

i2c.end()

---

i2c.read()

---

i2c.write()

---

i2c.requestfrom()

---

i2c.available()

Die Festlegung der zwei Anschlüsse für den Datenverkehr über die Datenleitungen SDA und SCL erfolgt z.B. mit *i2c.setup(0,2)* bei Benutzung dieser beiden Anschlüsse, die beim kleinen ESP01 zum Einsatz kommen und in BASIC vorkonfiguriert sind. Die Kommunikation mit einem teilnehmenden Gerät findet zwischen den Anweisungen *i2c.begin(adr)* – mit dem numerischen Wert der Geräteadresse – und *i2c.end()* statt. Dieser oftmals als Hexadezimalzahl aufgedruckte Wert kann bei einigen Geräten geändert werden, um mehrere dieser Geräte an denselben Leitungen zu betreiben. Die Datenübertragung erfolgt in beiden Richtungen (lesen/schreiben) ähnlich der Übertragung über die RX/TX Schnittstelle. Allerdings ist diese Übertragung synchron und nicht asynchron, wie bei den sonst verwendeten seriellen

Schnittstellen (UART). Dabei sendet *i2c.write(Byte)* ein einzelnes Byte zum Gerät. In der anderen Richtung kann mit *i2c.read()* ein Byte gelesen bzw. empfangen werden, wenn ein Zeichen vorliegt. Mit *i2c.available()* erfragt man die Anzahl der verfügbaren empfangenen Zeichen. Mit *i2c.request(adr,anzahl)* lassen sich mehrere Zeichen oder Bytes anfordern.

3.8.1 I2C-SCANNER

Zur Untersuchung der Adressen angeschlossener Geräte deren Adressen unsicher oder ganz unbekannt sind, dient folgender I2C-Scanner. Nach Anschluss des Geräts an die beiden Datenleitungen wie im Listing angegeben, erfolgt der Scan von 1 bis 127:

```
adr = 1
sda = 0
scl = 2
print “Scanne von “&adr&” bis 127”
print “SDA “&sda&” / SCL “&scl&” Moment …”
i2c.setup(sda,scl)
timer 10,[messen]
wait
[messen]
i2c.begin(adr)
if i2c.end() = “0” then
 a$ = “Finde Adresse 0x”
 a$ = a$ & upper(hex(adr,2))
 a$ = a$ & ” (” & adr & “)”
 print a$
endif
adr = adr+1
if adr > 127 then end
wait
```

Scanne von 1 bis 127

SDA 0 / SCL 2 Moment …

Done…

Demnach wäre die Adresse in hexadezimaler Schreibweise 3C, was dezimal dem Wert 60 entspricht. Dies ist insofern interessant als dass auf der Rückseite dieses OLED-Displays andere Angaben zu finden sind. Dieser Scanner wurde aus einem C/C++ Sketch übersetzt.

Abbildung 3-11: Veränderbare Adressen eines OLED-Displays

3.9 SPI VERBINDUNG

Eine weitere serielle Datenverbindung ist das 1987 entstandene SPI (Serial Peripheral Interface). Es verbindet verschiedene Bausteine mit synchronen seriellen Leitungen. Im Teil Anwendungen erfolgen Messungen mit einem SPI-Frequenzgenerator. Einige Anzeigen, wie das TFT-Display mit Touch, benutzen ebenfalls SPI. Diese serielle Übertragung ist üblicherweise schneller als I2C oder RX/TX, und kann in der Geschwindigkeit (Übertragungsfrequenz) angepasst werden. ESPBASIC erlaubt es durch die integrierten SPI-Befehle mit solchen Hardwareerweiterungen zu kommunizieren. Dieses schnelle Protokoll ist bei Mikrocontrollern mit Hardware-SPI-Unterstützung an feste Anschlüsse gebunden, die entsprechende Bezeichnungen tragen und am ESP8266 den folgenden GPIO zugeordnet sind:

GPIO14 (D5) SCK Takt

GPIO12 (D6) MISO Master *In* put, Slave Output (ESP-Eingang)

GPIO13 (D7) MOSI Master *Out* put, Slave Input (ESP-Ausgang)

Folgende Aufrufe stehen zur Verfügung.

spi.setup()

spi.byte()

spi.string()

spi.hex()

spi.setmode()

spi.setfrequency()

Mit *spi.setup(speed,mode,msb)* erfolgt die Initialisierung. Konkret reicht ein *spi.setup(1000000)* um z. B. mit der Datenrate 1MBit/s im Modus 0 und MSB_FIRST zu starten. Weitere Modi und die Reihenfolge der Datenbits (LSB/MSB) legen die optionalen Parameter fest. Gleichzeitiges Senden und Empfangen eines Bytes erfolgt mit *spi.byte(byte)* . Der Rückgabewert ist die Antwort, das als Parameter übergebene Byte wird gesendet. Ganze Zeichenketten sendet und überträgt *spi.string(string,len)* , wenn die beiden Parameter eine Zeichenkette und deren numerische Länge sind. Da Hardware oft mit Hexadezimalzahlen kommuniziert, gibt es der Einfachheit halber auch einen Befehl bzw. eine Funktion *spi.hex(hexstring,len)*. Ein Hex-String wäre „67ff00ea60“ mit 5 Bytes, die dann von links nach rechts gesendet werden. Auch hier ist der Rückgabewert die Antwort, allerdings als Hex-String. Bei nötiger Änderung des SPI-Modus nach der Initialisierung kann der Befehl *spi.setmode(mode)* mit 1 oder 0 umschalten. Die Frequenzänderung erfolgt entsprechend mit *spi.setfrequency(freq)* . Im nächsten Kapitel *Anwendungen* dieses Buches kommt die SPI-Ansteuerung im Zusammenhang mit dem Digital-Potentiometer MCP41010 auf der AD9833-Platine zum Einsatz. Die hier folgenden Zeilen stellen das Ausgangssignal an Pin PGA in 256 Stufen ein:

```
CSM = 2 ‘CS MCP41010
FSY = 16 ‘CS AD9833
DAT = 13 ‘HARD MOSI (12 is MISO)
CLK = 14 ‘HARD SPI
textbox Vu
button “OK”,[ok]
Vu = 127
spi.setup(100000)
wait
[ok]
gosub [gain]
wait
[gain] ‘MCP41010
data = 4352 or (Vu and 255)
```

```
io(po,CSM,0)
spi.byte(data / 256)
spi.byte(data)
io(po,CSM,1)
return
```

3.9.1 Fehlerhafte SPI-Modi in ESPBASIC

Wie in der Referenz angegeben ist, funktionieren in ESPBASIC nur die Modi 0 und 1. Das liegt daran, dass der ESP-Core in der Version 2.3.0 für die Arduino-IDE einen SPI-Fehler aufweist, mit der ESP8266BASIC 3.0 Branch 69 übersetzt wurde und als Binärdatei zur Verfügung steht. Der Quelltext zu ESPBASIC ist an dieser Stelle offensichtlich fehlerfrei. Dieser Bug hat zur Folge, dass bedauerlicherweise Geräte, die die Modi 2 oder 3 erwarten, mit Hardware-SPI nicht funktionieren. Der Modus legt fest bei welcher Takt-Flanke, bei aktivem Chip-Select CS, die einlaufenden seriellen Datenbits übernommen werden. Die folgende Darstellung versucht eine Übersicht.

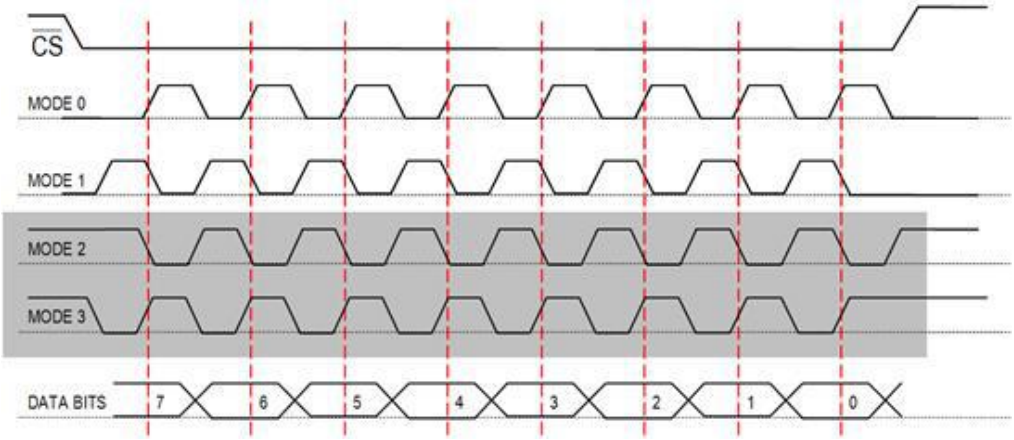


Abbildung 3-12: SPI-Taktflanken und SPI-Modi; Probleme bei Modus 2 und 3

Bei der Auswahl entsprechender Hardware ist ein Blick in das Datenblatt des verbendeten Bausteins hilfreich. Wenn der Takt im Ruhezustand auf High liegt, so macht das in ESPBASIC Probleme mit Hardware-SPI. Als Beispiel sei hier der Funktionsgenerator AD9833 genannt, der nur mit eigenem langsamen „Leitungsklappern“ in dieser BASIC-Version angesprochen werden kann, während das Digitalpotentiometer MCP41010 auf derselben Platine Hardware-SPI in ESPBASIC erlaubt. Auch der Frequenzgenerator AD9850 lässt sich problemlos und schnell über ESPBASIC und Hardware-SPI ansprechen. Schließlich sei noch bemerkt, dass Hardware-SPI und *IO* an denselben Anschlüssen problematisch sein kann, da eine *spi.free()*- Routine nicht vorhanden ist und erst ein *Reset* einige Leitungen wieder zurücksetzt.

4 ANWENDUNGEN

Dieser letzte Abschnitt greift die Inhalte der vorigen Kapitel auf und zeigt anhand von konkreten Anwendungen einige Lösungsmöglichkeiten mit ESPBASIC auf. Zu Beginn sind dies kleinere Beispiele mit wenigen Zeilen, zum Ende hin folgen aber auch Beispiele mit längeren Listings, die dann im Anhang aufgeführt sind, um den Gesamtzusammenhang nicht zu stark zu stören. Die Anwendungen reichen vom einfachen Diagramm im Browser bis zu aufwändigeren Messverfahren mit erweiterter Hardware.

4.1 MESSEN UND STEUERN IM BROWSER

Die Verschmelzung der allgemeinen und speziellen Befehle, sowie die Web-Elemente von ESPBASIC ermöglichen es mit relativ wenigen Zeilen eine für den Anwender übersichtliche Bedieneroberfläche zur Verfügung zu stellen. In diesem ersten Abschnitt der Anwendungen befinden sich kleinere Beispiele, die die verschiedenen Aspekte von ESPBASIC benutzen.

4.1.1 EINFACHE MESSTABELLE

Eine einfache Messtabelle der Analogwerte mit Zeitangaben kann direkt im Browserfenster erfolgen. Damit eine Kommazahl auch ein Komma enthält, wird der Punkt mit *replace()* ersetzt, wodurch die Darstellung in einem externen Programm auf einfachste Weise ermöglicht wird. Dieselben Messdaten können auf verschiedenen Geräten gleichzeitig angesehen und übernommen werden.

```
x = 0
meter v,0,1000
button “Stop”,[ende]
wprint “<hr>t/s   &nbspPIN ADC<hr>”
timer 1000,[machmal]
t0 = millis()
gosub [mach]
```

```
wait
[mach]
io(po,15,x)
x = not x
t = ((millis()-t0)/1000)
v = io(ai)
s = replace(str(t),".",",")
wprint s&"    "&v&"<br>"
return
[machmal]
gosub [mach]
wait
[ende]
End
```

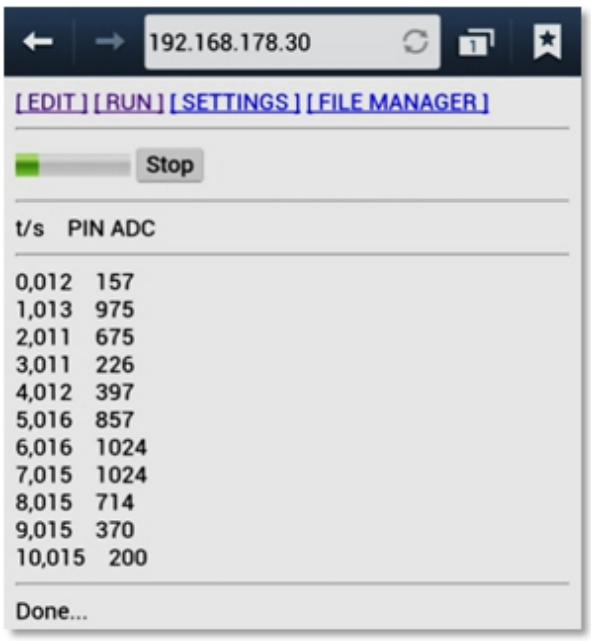


Abbildung 4-1: Einfache Tabelle im Browser zum kopieren

4.1.2 Messdiagramm aus Grundelementen

Mit den graphischen Grundelementen aus Abschnitt 2 lassen sich in ESPBASIC einfache Diagramme zeichnen. Die Werte 0 bis 5 sollen als Messpunkte erscheinen, wobei die X-Achse von 0 bis 10 und die Y-Achse von 0 bis 5 reichen sollen. Die Achseneinteilung enthält entsprechende Zahlenangaben und ein Gitter rundet das Erscheinungsbild ab. Die Zeichenfläche (Canvas) hat die Größe 320 x 320 Pixel mit einem Rand und

einem Diagrammbereich. Die Variablen *w* und *h* bestimmen die Größe der Zeichenfläche und mit *xw* , *yw* , *hw* , *ww* ist die Diagrammgröße innerhalb der Zeichenfläche festgelegt. Diese globalen Variablen benutzen die Unterprogramme *Punkt* und *Achsen* . *Punkt* berechnet aus den Diagrammeinheiten *x* und *y* die Pixeleinheiten *px* und *py* für die Zeichenposition.

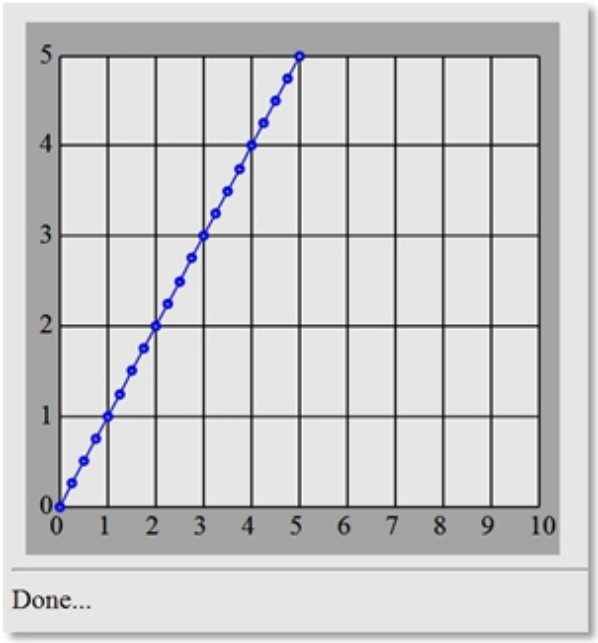


Abbildung 4-2: Einfaches Diagramm mit ESPBASIC-Grafik

Die Achsen mit Gitter und Beschriftung übernimmt das Unterprogramm *Achsen* , welches wiederum auf *Punkt* zugreift. Die Variablen *xmax* und *ymax* entsprechen den Endwerten der beiden Achsen in Diagrammeinheiten. In einer *For* -Schleife berechnet das Programm die lineare Funktion *y = x* und stellt den entsprechenden Punkt mittels der Grundfunktion *Circle* dar. Ab dem zweiten „Messwert“ wird eine Verbindung mittels *Line* zum vorigen Punkt gezogen, so dass die obige Darstellung entsteht.

```
w = 320
h = 320
xw = 20
yw = 20
ww = w - (2 * xw)
hw = h - (2 * yw) - 16
graphics w, h
rect 0, 0, w, h, 7
```

```
rect xw, yw, ww, hw, 15
gosub [achsen]
for x = 0 to 5 step 0.25
 y = x
 gosub [punkt]
 if x > 0 then
  line opx, opy, px, py, 9
 endif
 circle px, py, 3, 9
 circle px, py, 1, 15
 opx = px
 opy = py
next x
end
[punkt]
xmax = 10
ymax = 5
px = xw + (x / xmax * ww)
py = yw + hw - (y / ymax *hw)
return
[achsen]
xmax = 10
ymax = 5
for x = 0 to xmax
 y = 0
 gosub [punkt]
 text px - 6, py + 16, str(x), 0
 line px , py, px, yw, 0
next x
for y = 0 to ymax
 x = 0
 gosub [punkt]
 text px - 12, py + 4, str(y), 0
 line px, py, xw + ww, py, 0
next x
return
```

Bei der Beschäftigung mit den Grafikfunktionen stößt man schnell an Grenzen. Einerseits ist der Aufbau eher langsam und bisher frieren dynamische Diagramme nach kurzer Zeit ein. Ohne prozedurale Strukturen gestaltet sich diese Programmierung umständlich, da nur globale Variablen existieren. Diese und andere Gründe führen dazu, dass dieses Beispiel nur zeigen soll, dass es im Prinzip funktioniert. Weiter hinten wird jedoch auf JavaScript-Diagramme zurückgegriffen, die ganz ähnlich aufgebaut sind, aber viel schneller reagieren und auch bei dynamischen Messwerten mit entsprechenden Aktualisierungen keine Schwierigkeiten machen. Diese JavaScript-Dateien mit ihren Funktionen lassen sich aus dem ESP-Dateisystem oder aus dem Netz laden und müssen nur noch abgerufen werden.

Es sei wiederum bemerkt, dass ESPBASIC in C/C++ geschrieben ist, auf HTML und JavaScript aufsetzt, um die Oberfläche zu gestalten, graphische Befehle ebenfalls in JavaScript umsetzten muss. Dass dabei Reibungsverluste entstehen sollte nicht verwundern. Die Möglichkeit der Einbindung von JavaScript macht ESPBASIC zu einem mächtigen Werkzeug.

4.1.3 STEUERN IM BROWSER

Steuern bedeutet oft nur Schalten. Im einfachsten Fall ist das dann das Ein- und Ausschalten von Digitalausgängen.

Abbildung 4-3: Ein- und Ausgänge im Browser bedienen

Bei dem hier benutzten Exemplar Witty Cloud ist eine RGB-LED verbaut, die die drei Pins 15, 12, 13 - in dieser Reihenfolge - belegt. Mit drei *button* -Elementen erfolgt die Programmverzweigung, oder anders ausgedrückt: Drei *button* -Elemente lösen jeweils ein Ereignis aus, was dazu benutzt wird, den entsprechenden Ausgang zu schalten. Das optische Feedback im Browser übernehmen drei *meter* -Elemente, wie weiter oben im Taster-Beispiel mit dem Interrupt. Damit auch wirklich der aktuelle Zustand erscheint, erfolgt die Abfrage über *laststat* . Da Übertragungen per Socket bzw. TCP/IP nicht immer gleich reagieren, erfolgt die Kontrolle doch wieder über eine Art Polling, also als ständige Abfrage, allerdings per Timer. Aus optischen Gründen wird hier rechts neben dem *meter* -Element ein *dropdown* -Element platziert. Dieses Bedienelement wird als 0/1-Anzeige missbraucht und könnte den Benutzer verwirren, da es nicht entsprechend reagiert - geschaltet wird nur per Taster.

'RGB SWITCH

x = 0

blau=-1

grun=-1

rot=-1

```
timer 100,[mach]
button “R”,[rot]
meter r,0,1
dropdown r, “0,1”
wprint “<br>”
button “B”,[blau]
meter b,0,1
dropdown b, “0,1”
wprint “<br>”
button “G”,[grun]
meter g,0,1
dropdown g, “0,1”
wprint “<br>”
meter v,0,1000
button “Exit”,[ende]
wait
[mach]
io(po,2,x)
x  = not x
v = io(ai)
r = abs(io(laststat,15))
b = abs(io(laststat,13))
g = abs(io(laststat,12))
wait
[rot]
io(po,15,rot)
rot = not rot
wait
[blau]
io(po,13,blau)
blau = not blau
wait
[grun]
io(po,12,grun)
grun = not grun
wait
```

[ende]

io(po,2,-1)

cls

end

Um alles untereinander anzuordnen erfolgt ein *HTML-
* zwischen den Elementen. Am Ende wird die blaue, hellere LED an Pin 2 noch ausgeschaltet und der Bildschirm gelöscht. Durch das Blinken der kleinen blauen LED ist das Pulsieren der Helligkeit im analogen Balken (LED-Abfrage) neben dem Exit-Taster deutlich zu erkennen, wenn eine helle Fläche als Reflektor benutzt wird. Auch diese Oberfläche kann von verschiedenen Geräten betrachtet und bedient werden.

4.1.4 Analoges Steuern

Die Ausgänge des ESP sind PWM-fähig, können also mittels Pulsbreiten-Modulation quasi-analoge Ausgaben machen. Der arithmetische Mittelwert eines Rechtecksignals entspricht seinem Tastverhältnis und so leuchten die drei LED entsprechend hell, je nach Ausgabewert zwischen 0 (aus) und 1023 (an).

Abbildung 4-4: Analoge Bedienelemente und deren Anwendung

Als Bedienelement stellt ESPBASIC einen Schieber (*slider*) zur Verfügung. Die Programmierung ist genau so einfach wie bei einem *meter* -Element, nur dass durch das Schieben durch den Anwender die zugewiesene Variable im angegebenen Bereich verändert wird. Wird die Variable im Programm verändert, so bewegt sich der Schieber entsprechend. Auf diese Art können nun mit folgendem Quelltext die drei LED

an den Ausgängen 15, 12 und 13 mit 1023 Helligkeitsstufen gesteuert werden.

```
‘rgbSLIDE
wprint “R “
textbox r
slider r,0,1023
wprint “<br>G “
textbox g
slider g,0, 1023
wprint “<br>B “
textbox b
slider b,0, 1023
wprint “<br>”
x = 0
T = 500
wprint “A “
textbox v
wprint “     ”
meter v,0,1023
timer T,[mach]
wprint “<br><br>Intervall/ms: “
dropdown bla, “10,100,500,1000,10000”
button “Exit”,[ende]
wait
[mach]
if or <> r then io(pwo,15,r)
if og <> g then io(pwo,12,g)
if ob <> b then io(pwo,13,b)
io(po,2,x)
x  = not x
v = io(ai)
r = abs(io(laststat,15))
b = abs(io(laststat,13))
g = abs(io(laststat,12))
or = r
```

```
og = g
ob = b
T = val(bla)
timer T,[mach]
wait
[ende]
io(po,2,-1)
cls
end
```

Damit die Angaben auch stimmen, erfolgt per Timer eine Kontrollabfrage der Ausgangszustände. Das Abfrage-Intervall kann per *dropdown* -Element über die Variable *bla* geändert werden.

4.2 Messen und Darstellen mit JavaScript

In diesem Abschnitt kommt eine kleine JavaScript-Bibliothek zum Einsatz, die aus einer Pascal-Portierung entstand und als *bt93.js* im Jahr 2012, mit Auftritt von HTML5 mit seinem Canvas, erstmals auf ***www.hjberndt.de/soft/canbt93.html*** zum Einsatz kam. In diesem Pfad liegt auch die Bibliothek ***www.hjberndt.de/soft/bt93.js*** und ist oder war im Browser über diesen Link erreichbar. Sie dient der Erzeugung einfacher Diagramme aus Messdaten und steht dabei als Platzhalter anderer im Netz verfügbarer Bibliotheken.

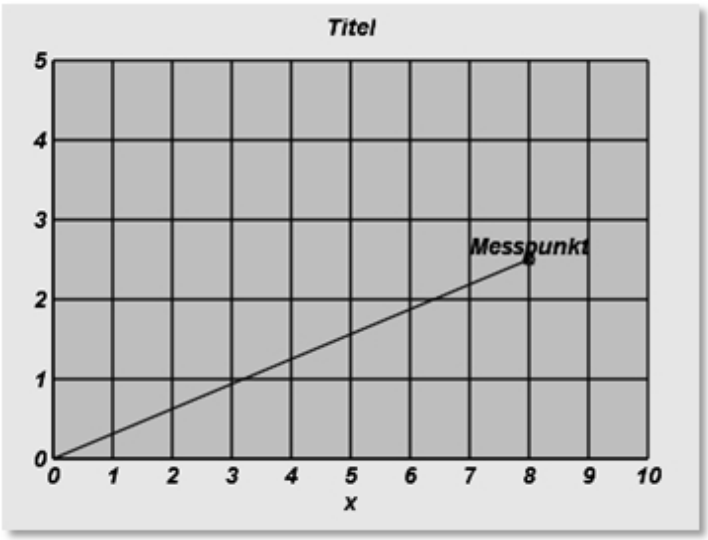


Abbildung 4-5: Einfaches Diagramm mit JavaScript

Im Anhang befindet sich eine Kurzreferenz der wenigen Routinen und Konstanten, um die Benutzung oder auch Änderung zu ermöglichen. JavaScript als Sprache ist nicht Gegenstand dieses Buches, die Anwendung im Umfeld von ESPBASIC bietet sich wegen dessen Unterstützung dieser Sprache jedoch an. Das oben dargestellte Diagramm entsteht, wenn eine HTML-Seite den folgenden Aufbau hat:

```
<!DOCTYPE html><html><body><br>

<canvas id="myCanvas" width="480" height="320" >

</canvas>
```

```
<script src="bt93.js" type="text/javascript">
</script>
<script type="text/javascript">
Grafik(AN);
farbe=WEISS;cls();
farbe=HELLGRAU;     hintergrund();
farbe=SCHWARZ;      xachse="x";yachse="";
Diagramm(0,10,0,0,5,1);
marktyp=KREUZ|KREIS;
DiaPunkt(8.0,2.5);
DiaLinie(0,0,8,2.5);
DiaText(8,2.5,"Messpunkt");
</script>
</body>
</html>
```

Es ist eine Mischung aus HTML und JavaScript und enthält im Wesentlichen die

- Erzeugung einer Zeichenfläche (Canvas)
- Einbindung der JavaScript-Bibliothek (bt93.js)
- Erzeugung der Grafik

Die Grafik besteht aus einem Diagramm und einem Messpunkt und entsteht durch

- Initialisierung der Grafik
- Farbgestaltung von Zeichenfläche und Diagrammfläche
- Zeichnen des Diagrammgitters und Achsen
- Anpassung der Messpunktdarstellung
- Darstellung von Messpunkt, Linie und Text in Diagrammeinheiten

ESPBASIC kann mittels *htlm* oder *wprint* auch HTML schreiben oder „printen“, sodass dieser Seitenquelltext jetzt

aus BASIC heraus erzeugt wird (vgl. Abschnitt 2). Nach entsprechenden Kopiervorgängen kann das Listing wie folgt aussehen:

```
html |<canvas id="myCanvas" width="480" height="320">|
html |</canvas>|
html |<script src="http://hjberndt.de/soft/bt93.js"|
html |type="text/javascript"></script>|
html |<script type="text/javascript">|
html |Grafik(AN);|
html |farbe=WEISS;cls();|
html |farbe=HELLGRAU;  hintergrund();|
html |farbe=SCHWARZ;  xachse="x";yachse="";|
html |Diagramm(0,10,0,0,5,1);|
html |marktyp=KREUZ|KREIS;|
html |DiaPunkt(8.0,2.5);|
html |DiaLinie(0,0,8,2.5);|
html |DiaText(8,2.5,"Messpunkt");|
html |</script>|
```

Da hier nur die Bibliothek aus dem Netz kommt, jedoch nicht die Seite selber, ist die Angabe der gesamten Adresse nach *src* erforderlich. Alternativ kann die Bibliothek auch aus dem Dateisystem von ESPBASIC geladen werden. Denkbar wäre auch das Hochladen der HTML-Seite ins BASIC-Dateisystem.

Nach dem Start dieses „BASIC-Programms" erfolgt die Darstellung, wie oben angegeben.

4.2.1 Dynamisches Messen im Diagramm

Mit dem Zusammenspiel zwischen ESPBASIC, mit seinen Hardware-Funktionen, und der JavaScript Unterstützung ist es möglich Messdaten zwischen den beiden Programmsprachen zu „teilen": BASIC misst und JavaScript stellt dar. Als Grundlage dient der Quelltext, wie er unter

http://hjberndt.de/soft/BTLAUF.html

als "NoTrigger" abgerufen werden kann. Mit anderer Farbgestaltung, aber überwiegend gleichem Quelltext entsteht eine schnelle Messwertdarstellung des Analogeingangs vom

ESP8266F mit angeschlossenem LDR. Die Reflexion der pulsierenden roten LED in der Messroutine von 100 ms ist deutlich sichtbar. Danach wird durch Drehung der Anordnung Tageslicht registriert.

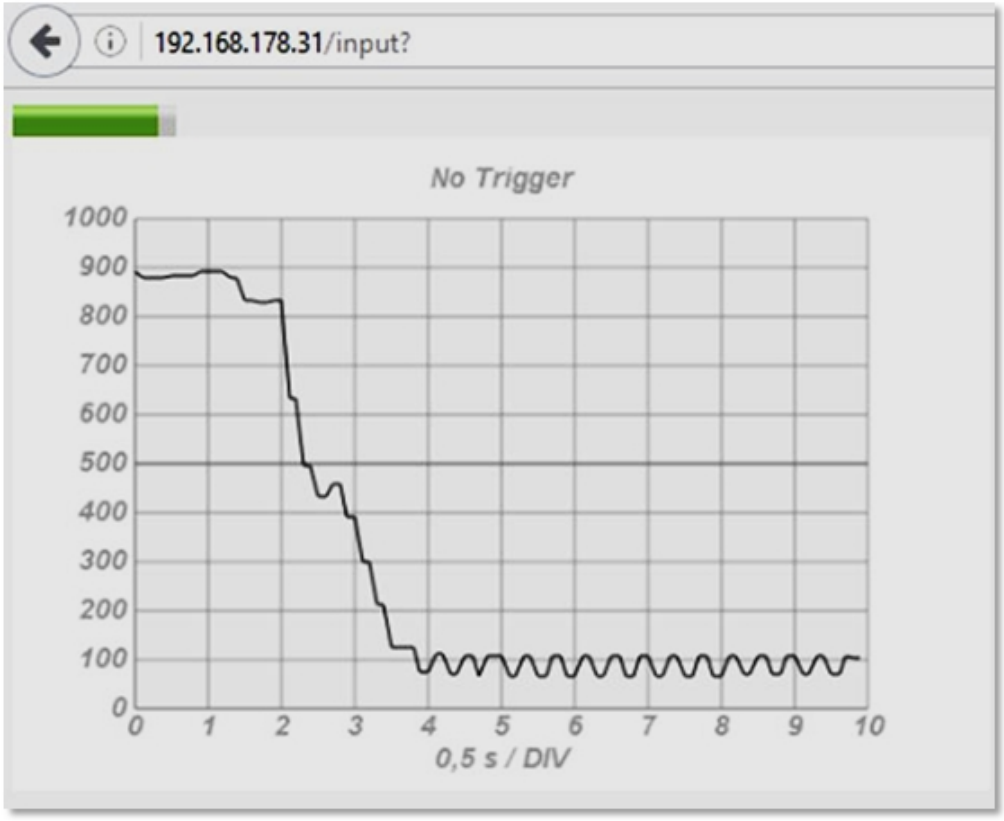


Abbildung 4-6: Scnelle Messgrafik in JavaScript und ESPBASIC

Diese Adaption bettet den HTML/JavaScript-Quelltext komplett in BASIC ein, was speichertechnisch bei 1 MB gerade noch so eben passt. Auf Leerzeichen wird aus Platzgründen verzichtet, wodurch das Listing nicht sehr lesefreundlich erscheint. Wegen des Layouts erfolgen teilweise Zeilenumbrüche, die im Listing nur eine Zeile darstellen und gegebenenfalls zu Fehlern führen. Die Bibliothek kommt diesmal direkt vom ESP-Dateisystem, hochgeladen mittels ESPBASIC-FileManager.

```
x = 0
y = 88’NoTrigger
meter y,0,1000
idy = htmlid()
timer 100,[messen]
html |<html><body bgcolor=“whitesmoke”><br>|
```

```
html |<canvas id="myCanvas" width=480 height=320|
html | style="border:0px solid #c3c3c3"></canvas>|
html |<script src="/file?file=bt93.js"|
html |type="text/javascript"></script>|
html |<script type="text/javascript">|
html |var name_element = document.getElementById("| & htmlvar(idy) & |");|
html |const NMAX=100;var filled=false;var tmax=10;|
html |var TabY = new Array(NMAX);var TabX = new Array(NMAX);|
html |function Messen(){var i,x,y,t,now=new Date();|
html |if(filled) {y=TabY[NMAX-1];  for(i=NMAX-1;i>0;i—) TabY[i]=TabY[i-
1]; TabY[0]=name_element.value;}|
html |else {for(i=0;i<NMAX;i++) {x=i/NMAX*tmax;
y=name_element.value;TabX[i]=x;|
html |TabY[i]=y} filled=true;}}|
html |function Anzeigen(){var i; Grafik(AN); Messen();
yachse="";xachse="0,5 s / DIV";titel="No Trigger";|
html |farbe ="White";cls();farbe="whitesmoke"; hintergrund();
farbe="DarkGrey" ; Diagramm(0,tmax,0,0,1000,0); farbe="gray"; |
html |DiaLinie(0,500,tmax,500); farbe="black "; |
html |for(i=0;i<NMAX-1;i++)
DiaLinie(TabX[i],TabY[i],TabX[i+1],TabY[i+1]);}|
html |window.setInterval("Anzeigen()", 50);</script><br></body></html>|
wait
[messen]
x = not x
io(po,15,x)
y = io(ai)
wait
```

Im Original wird ein Sinus theoretisch berechnet und dem Array zugeführt. Hier oben erfolgt die Messdatenerfassung und Darstellung durch *io(ai)* in der Messroutine, dem Zuweisen des Wertes an das *meter* -Element, der Weiterreichung der *Id* dieses HTML-Elements an *JavaScript* , das dann über die *Id* den Wert abruft und mit *TabY[0] = name_element.value* in die Messtabelle vorne einträgt, die von der Anzeige-Routine gezeichnet wird. Die zeitliche Darstellung der Kurve wird im Listing an drei Stellen beeinflusst. Die Routine *messen* wird hier über einen *timer* alle

100 ms aufgerufen, das Diagramm zeigt 10 Skalenteile und *JavaScript* zeichnet die Ausgabe - oder versucht es zumindest - alle 50 ms.

4.2.2 YT-SCHREIBER

Für langsamere Vorgänge kann ein TY-Schreiber nachgebildet werden.

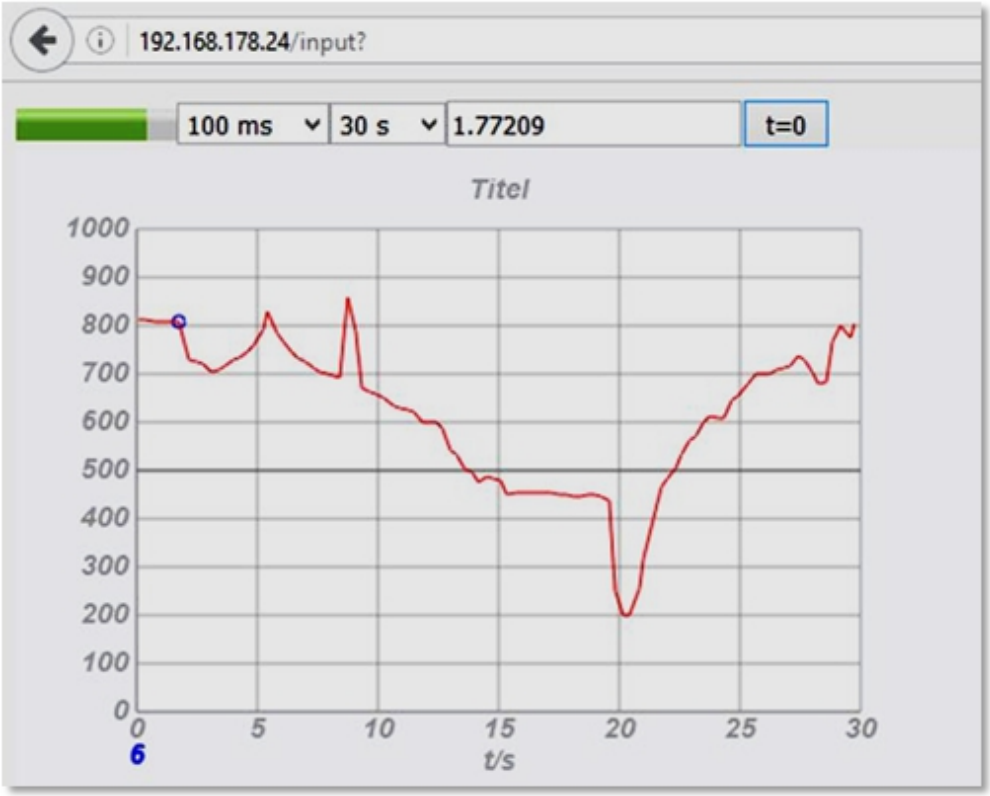


Abbildung 4-7: : TY-Schreiber für langsamere Vorgänge

Das Beispiel misst 30 Sekunden alle 0,1 s und trägt das Ergebnis in die Tabelle mit 100 Plätzen ein. Die Speicherposition - hier die 6 - errechnet sich aus Messdauer und Intervall und wird blau angezeigt. Der Zeitpunkt steht im rechten Textfeld. Mit 100 Messwerten und etwas Benutzeroberfläche gelangt auch diese Version schnell an Speicherplatz-Grenzen.

Intervall und Messdauer sind frei einstellbar, um die unterschiedliche Hardware- und Grafikfähigkeit von Endgeräten zu berücksichtigen, aber es ist nur eine Spielwiese zum Experimentieren. Soll das Diagramm bei jedem Neustart gelöscht werden, so ist als Endwert der *For* -Schleife bei der

Anzeige die Konstante *NMAX* durch die Variable *ix* zu ersetzen. Das Listing befindet sich im Anhang 5.3.2.

4.3 Messen und Steuern mit Excel und Word

Mit Hilfe von ESPBASIC können Messungen in Excel oder Word direkt per WiFi ausgeführt, dargestellt und eventuell ausgewertet werden:

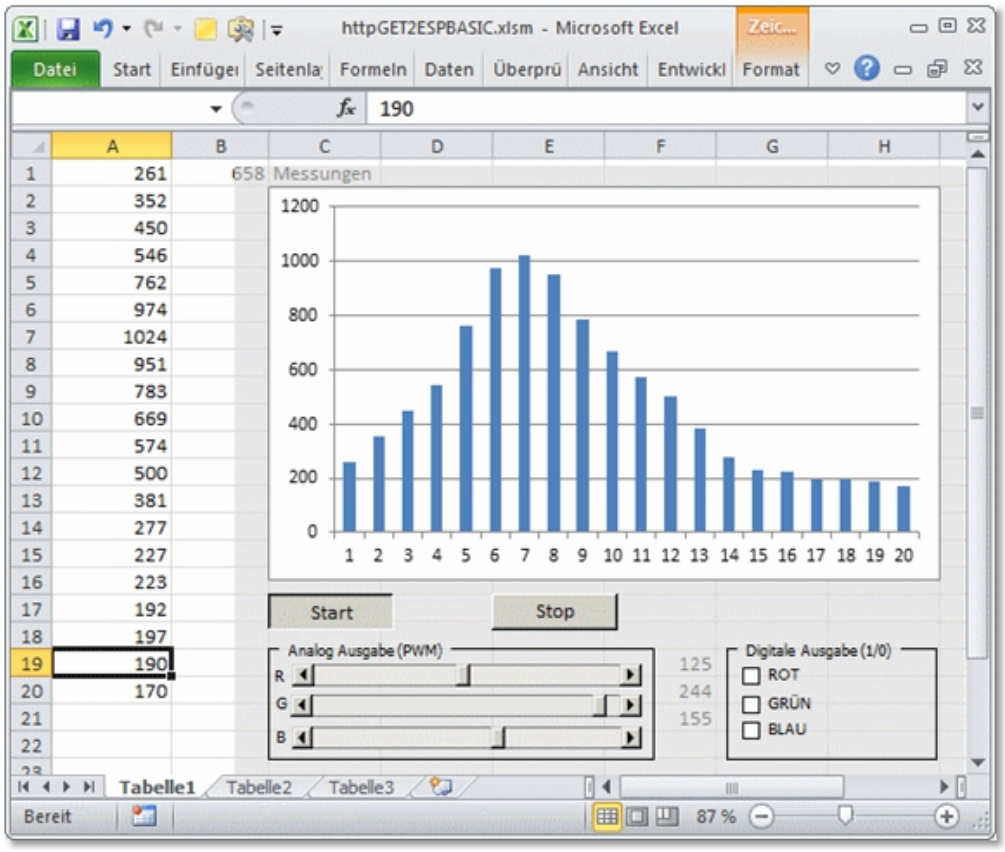


Abbildung 4-8: ESPBASIC kommuniziert mit Excel ganz ohne Zusatzsoftware

Das Buch „Messen Steuern Regeln mit Smartphone und Tablet“ [3] warf die Frage auf, ob es nicht möglich sei, ohne Umwege und DLL’s aus Office heraus direkt mit dem ESP8266 zu kommunizieren. Im obigen Buch ist das nur über Umwege dargestellt und durchgeführt worden, da Makros in VBA keine TCP/IP-Sockets unterstützen. Bei der Auseinandersetzung mit dem Problem und genauerem Studium der Referenz zu ESPBASIC fiel auf, dass beide Sprachen über die Möglichkeit verfügen, den sogenannten

http-GET-Befehl zu benutzen. Mit dieser Funktion kann man Webseiten aus dem Netz laden und deren Inhalt auswerten. ESPBASIC nennt dieses Kommando *wget* (siehe oben), welches in VBA nicht direkt verfügbar ist, jedoch mit wenigen VBA-Zeilen erzeugt werden kann. Damit lassen sich Daten vom Browser-Fenster, durch ESPBASIC erzeugt, mit einem Excel-Makro abholen.

4.3.1 ESP8266 ALS HTTP-SERVER

Die Kommunikation mit dem ESPBASIC erfolgt über die Adresszeile, wie im Abschnitt 2.6 erläutert. Der Übertragungsweg ist dann wie folgt:

Aus einem Microsoft-Office-Programm, wie z.B. Excel wird das „VBA-wget“ aufgerufen, wobei in der URL dann die gewünschte Aktion enthalten ist. ESPBASIC wertet diese Daten nach der Zeichenfolge *msg?* in der Anfrage aus und reagiert entsprechend mit der gewünschten Ausgabe an einem Pin oder mit der Rückgabe eines Messwertes, der mit *msgReturn* ins Browser-Fenster geschrieben wird. Die VBA-*wget* -Funktion erhält diesen Messwert in Form einer Zeichenkette, um den Wert weiter zu bearbeiten. Auf Seiten von ESPBASIC sieht das Programm dann wie folgt aus:

```
msgbranch [mybranch]
wait
[mybranch]
pinNo = val(msgget(“pin”))
pinStatus = val(msgget(“stat”))
pinAction = msgget(“action”)
MyReturnMsg = “OK”
if pinAction == “po” then
   io(po,pinNo,pinStatus)
end if
if pinAction == “pi” then
   MyReturnMsg = io(pi,pinNo)
end if
if pinAction == “pwo” then
   io(pwo,pinNo,pinStatus)
```

```
end if
if pinAction == “pwi” then
   MyReturnMsg = io(pwi,pinNo)
end if
if pinAction == “ai” then
   MyReturnMsg = io(ai)
end if
msgreturn MyReturnMsg
wait
```

Dazu wurde das Beispiel von

http://www.esp8266basic.com/examples.html

(advanced msg example) nur geringfügig angepasst. Insbesondere finden in der Modifikation Ausgaben im Browser-Fenster nur noch mit *msgreturn* statt, da sonst bei Dauerbetrieb möglicherweise Probleme auftreten. Über die Schlüsselwörter *action* , *pin* , *stat* verzweigt die Auswertung zur Portausgabe *po* , Porteingabe *pi* , Portausgabe *pwo* , Porteingabe *pwi* und der Analogabfrage *ai* des einzigen Analogeingangs. Die beiden anderen Parameter erklären sich von selbst anhand eines Beispiels. Soll aus Excel heraus der Pin 2 (LED) am ESP8266 angeschaltet werden, so lautet die von VBA-Excel zu übertragende Zeichenkette der VBA-*wget* -Funktion

http://192.168.4.1/msg?pin=2&stat=1&action=po **,**

wenn der ESP als eigener Access-Point in der Luft ist und der Excel-Rechner in seinem Netzwerk angemeldet ist. Das sind jede Menge Daten im Vergleich zu TCP/IP oder RS232, aber die Verbindung klappt drahtlos, ohne Treiberprobleme und auch ohne Internet und Router. Hier wird demnach die ESPBASIC-Anweisung *io(po,15,1*) ausgeführt; also die LED angeschaltet. Wird in der Zeile *stat=1* in *stat=0* geändert, schaltet die LED aus. Damit können über die Adresszeile eines Browsers alle IO-Kommandos ausgelöst werden. Um den Analogwert des Fotowiderstandes am Witty Cloud zu erhalten reicht:

http://192.168.4.1/msg?action=ai

und der ESP sendet das Ergebnis von *io(ai)* .

4.3.2 Excel und Word VBA-Einbindung

Nach diesem Schema lassen sich in VBA eigene Aufrufe programmieren, um die gewünschten Daten zu erhalten oder zu senden. Die VBA-*wget* -Funktion wird als *Function ESP(ByVal cmd$)* programmiert. Sie steht am Anfang eines neuen VBA-Moduls im VBA-Editor und bekommt die Kommandozeile als String übergeben.

Achtung! Hier folgt Visual Basic für Excel & Word, kein ESPBASIC.

```
Function ESP ( ByVal Cmd$)

Set HTTP = CreateObject( "WinHttp.WinHttpRequest.5.1" )

    HTTP.Open "GET" , "http://192.168.4.1/msg?" + Cmd$

  HTTP.Send

    ESP = HTTP.ResponseText

    Set HTTP = Nothing

End Function
```

Diese Funktion führt die http-Anfrage durch und entspricht so etwa dem *wget* -Befehl. Mit der Zuweisung *ESP = http.ResponseText* liefert die Funktion die empfangenen Daten dem Aufrufer. Ein Test könnte im Direktbereich des VBA-Editors stattfinden in dem dort

Print ESP(action=ai)

mit der Eingabetaste angefordert wird. VBA sollte den Analogwert des ESP8266 darunter ausgeben.

Sollen Messwerte direkt in ein Tabellenblatt oder Dokument geschrieben werden, so sind kurze eigene Routinen sinnvoll. Zur Steuerung der roten LED am Witty Cloud könne ein solcher Befehl (als VBA sub) zum Beispiel „RT“ für die Farbe Rot lauten. RT 1 schaltet dann ein und RT 0 schaltet aus. Als VBA-Sub schreibt sich dass dann:

```
Sub RT ( ByVal an)

ESP ( "action=po&pin=15&stat=" + Str$(an))

End Sub
```

Im Direktbereich reichen nun diese Kommandos, um mit ESPBASIC und der Hardware zu „sprechen“. Das Buch „Messen, Steuern und Regeln mit Word und Excel“ [4] hat VBA-Makros als Hauptthema. Zum Zeitpunkt der Erstausgabe war WiFi überwiegend unbekannt und das Internet eine Nebenerscheinung und die Schnittstelle war die serielle RS232. Mit HTTP klappt das nun aber auch wieder mit dem ESP8288 und seinem einfach zugänglichen ESPBASIC. Anhand der gegebenen Kurzbeispiele entsteht hier ein Gesamtmakro, welches „as-it-is“ hier dargestellt wird. Auch Steuerelemente von Excel sind im Makro berücksichtigt, damit das oben dargestellte Tabellenblatt entsprechend funktioniert. Eine tiefere Auseinandersetzung mit VBA-Makros ist hier aus Platzgründen nicht vorgesehen.

```
'http://www.esp8266basic.com/msg-url-advanced.html

'RGBLED ESP WITTY CLOUD RT = 15, BL = 13, GN = 12

Function ESP ( ByVal Cmd$)

Set HTTP = CreateObject( "WinHttp.WinHttpRequest.5.1" )

   HTTP.Open "GET" , "http://192.168.4.1/msg?" + Cmd$

   HTTP.Send

   ESP = HTTP.ResponseText

   Set HTTP = Nothing
```

```
End Function

Sub Delay ( ByVal ms)

t = Timer

While Timer < (t + ms / 1000 )

  DoEvents

Wend

End Sub

Sub BL ( ByVal an)

ESP ( "action=po&pin=13&stat=" + Str$(an))

End Sub

Sub RT ( ByVal an)

ESP ( "action=po&pin=15&stat=" + Str$(an))

End Sub

Sub GN ( ByVal an)

ESP ( "action=po&pin=12&stat=" + Str$(an))

End Sub

Sub DOUT ( ByVal pin, ByVal an)

ESP ( "action=po&pin=" + Str$(pin) + "&stat=" +
Str$(an))

End Sub

Sub AOUT ( ByVal pin%, ByVal pwm%)

ESP ( "action=pwo&pin=" + Str$(pin) + "&stat=" +
Str$(pwm))

End Sub
```

```
Sub ReadLDR ()
Range( "A:A" ).ClearContents
While True
 For i = 1 To 20
  x = ESP( "action=ai" )
  Cells(i, 1 ).Select
  Cells(i, 1 ) = x
  Range( "b1" ) = n
  Delay 200
  n = n + 1
 Next i
Wend
End Sub
Sub BildlaufleisteRT_Change ()
   AOUT 15 , Range( "F19" )
End Sub
Sub BildlaufleisteGN_Change ()
   AOUT 12 , Range( "F20" )
End Sub
Sub BildlaufleisteBL_Change ()
   AOUT 13 , Range( "F21" )
End Sub
Sub CheckboxRT_Change ()
```

```
If Range( "i19" ) Then an = 1 Else an = 0

DOUT 15 , an

End Sub

Sub CheckboxGN_Change ()

If Range( "i20" ) Then an = 1 Else an = 0

DOUT 12 , an

End Sub

Sub CheckboxBL_Change ()

If Range( "i21" ) Then an = 1 Else an = 0

DOUT 13 , an

End Sub

Sub Ende ()

End

End Sub
```

Visual Basic meets ESPBASIC: Messen und Steuern im Tabellenblatt von Excel. Ganz ohne DLL erfolgt die Verbindung zum ESP8266 über WiFi/HTTP. Die Messroutine erfragt alle 0,2 Sekunden den Analogwert. Mit den Schiebereglern kann die Helligkeit der drei Farben quasi analog gesteuert werden. Die Checkboxen schalten an und aus. Die Elemente aus der Formularbox erfordern entsprechende Zuweisungen, wie in VBA/Excel üblich. Im Wesentlichen hat sich nur der Übertragungsweg geändert. Dieser Beitrag wurde in Teilen als Nachtrag zum Buch „Messen Steuern Regeln mit Smartphone und Tablet“ [3] bereits 2018 unter

http://hjberndt.de/soft/esp2excelbas.html

veröffentlicht. Einmal ist dieses Tabellenblatt dort als Zip verlinkt. Der Link ist an ein kleines, fettes l von Excel geheftet. Oder direkt über:

http://hjberndt.de/soft/GET2ESPBASICnetz.zip

Müssen kurz in Word Messwerte live in das Dokument geschrieben werden, so ändert sich lediglich die Messroutine wegen der direkten Ausgabe zu:

```
Sub ReadLDR()

For i = 1 To 20

  DoEvents

  x = ESP("action=ai")

  Selection.TypeParagraph

  Selection.TypeText Text:=i & vbTab & x

  Selection.TypeParagraph

  Delay 200

Next i

End Sub
```

4.4 RS232 - WiFi-Konverter für alte Hardware

Ein ESP8266 ist ein Seriell-zu-WiFi-Wandler und mit dieser Eigenschaft kann drahtgebundene Hardware in drahtlose Hardware gewandelt werden.

Abbildung 4-9: Altes RS232-Inteface mit Sub-D9-Buchse (female) an der Seite

In diesem Abschnitt soll ein konkretes, älteres Gerät mit 9-poliger Sub D-Buchse (female), welches normalerweise mit einem RS232-Kabel in eine serielle Schnittstelle am PC (COM) mit Sub D-Stecker (male) gesteckt wird, seine Daten und Steuersignale drahtlos über WiFi erhalten und somit das Kabel überflüssig machen. Eine Veränderung oder Öffnung des Geräts ist nicht vorgesehen, kann aber bei Bedarf erfolgen. Der ESP8266 mit seinem ESPBASIC tritt dabei als Konverter oder Vermittler auf.

Aufgrund der unterschiedlichen Spannungspegel ist eine Anpassung mittels Hardware erforderlich. Arbeiteten und arbeiten RS232-Schnittstellen mit etwa 10 Volt und umgekehrten Logikpegel, so benutzt TTL-Logik einen 5 Volt Pegel, während der ESP8266 3,3 Volt-Pegel verarbeitet. Für diese Zwecke gibt es preiswerte bidirektionale Pegel-Shifter, die zwischen den beiden niedrigen Pegeln als Hardware vermitteln. Ein MAX3232-Baustein generiert aus seiner Betriebsspannung die für die umgekehrte Logik erforderlichen 10 Volt, damit die dafür konstruierten Geräte ebenfalls den richtigen Spannungspegel erhalten. Konverter mit MAX3232 gibt es auch mit Sub D-Stecker(!), die dann direkt in das betreffende Gerät passen, genauso wie das PC-Kabel. Hier wird also das Kabel durch MAX2323 und ESP8266 ersetzt; der PC kommuniziert dann über WiFi mit dem Gerät und das mittels ESPBASIC.

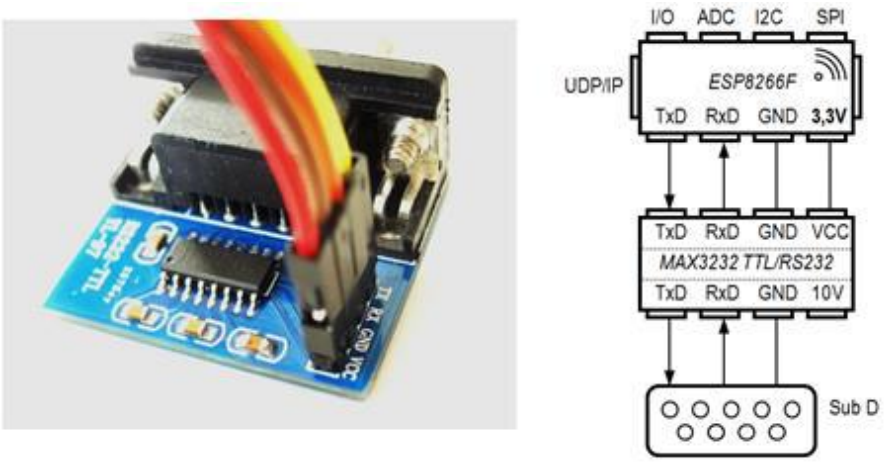


Abbildung 4-10: MAX3232 mit Sub-D9-Stecker (male) und schematischer Aufbau

4.4.1 Konverter – RS232 zu TTL-Serial

In Kombination mit dem ESP8266F entsteht der schematische Aufbau des Konverters bzw. Adapters. Der Sub D-Stecker im Gerät mit seinen 9 Pins liefert die RS232-RX/TX-Signale an den MAX3232, der diese in 3,3 Volt RX/TX-Pegel wandelt. Beim Witty Cloud ist Vorsicht geboten, da der dortige Pin VCC an 5 Volt liegt! Gleich nebenan ist eine kleine Platinen-Bohrung mit 3,3 V-Pegel, so dass der MAX3232 mit der richtigen Spannung versorgt werden kann.

Levelshifter sind somit bei dieser Lösung überflüssig. Das TX-Signal läuft über Sub-D und MAX zum TX-Anschluss des ESP. Die RX-Verbindung kommt entsprechend an RX des ESP. Wenn die Verbindung steht, kann der erste Test beginnen.

4.4.2 Digitale Ausgaben

Die konkrete Testsituation erfolgt mit dem oben abgebildeten älteren Interface, wie es auch an Schulen zum Einsatz kam/kommt und wegen der „COM-Schnittstelle“ möglicherweise auf der Entsorgungsliste landet. Tatsächlich kann damit auch ohne COM-Anschluss gemessen und gesteuert werden und das sogar mit dem auch in Schulen schon gesichteten Mobilfunkgeräten alias Smartphones. Das Interface erhält seine 12 Volt Spannung Versorgung unverändert über ein entsprechendes Netzteil. Das serielle Kabel wird durch die MAX3232-Platine ersetzt und mit dem SUB-D-Stecker mit dem Interface verbunden. Dabei kann es

nötig sein, die Verschraubung der Steckverbindung einseitig zu entfernen. Vier Leitungen verbinden die MAX-Platine mit dem ESP8266 wie folgt:

MAX	ESP
RX	RX
TX	TX
GND	GND
VCC	3,3V (!)

Das Gerät verfügt über zwei analoge Eingänge mit 8 Bit-Auflösungen sowie 8 Digitalausgänge und 8 Digitaleingänge. Es ist Software kompatibel zu einem *Camface* der Firma Phywe aus derselben Zeit. Die Steuerung erfolgt durch Senden und Empfangen von einzelnen Bytes. Die Digitalausgänge reagieren auf das Steuerbyte 81 und anschließendem Datenbyte, wodurch die Leuchtdioden und die Digitalausgänge entsprechend dem Datenbyte gesetzt werden. Bei der Übertragung der beiden Bytes 81 und 85 (nicht die Zahl) erscheint an den 8 Ausgängen die entsprechende Binärdarstellung als Muster 01010101. Ein erstes Testprogramm sendet darum Timer gesteuert diese Sequenz in entsprechender Baudrate, um bei Erfolg mit der binären Negation der 85 (170 - 10101010) ein Blinkmuster zu erhalten.

```
baudrate 19200
timer 500,[tm]
x = 85
[tm]
serialprint chr(81)
serialprint chr(x)
x = not x and 255
wait
```

4.4.3 Analoge und digitale Eingaben

Nach erfolgreicher unidirektionaler Ausgabe an den Digitalausgängen folgt der Test der Eingänge. Das Byte, welches den Bits der Digitaleingänge entspricht, liefert diese Hardware als Antwortbyte auf ein seriell empfangenes Byte mit dem Wert 63.

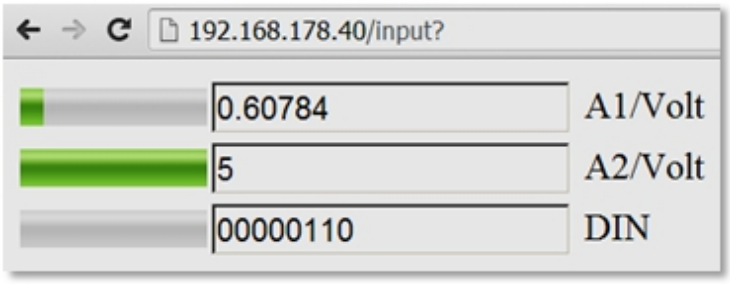


Abbildung 4-11: Messen in Echtzeit über RS232/WiFi

Die beiden Analogeingänge reagieren auf 60 und 58 mit dem Wert des A/D-Wandlers im Bereich 0 bis 255, was 0 bis 5 Volt am Eingang entspricht. Im Folgenden, erweiterten Listing wartet ein *serialbranch* auf die Antwort eines gesendetes Byte mit dem Wert 60 im Timer-Intervall, um den Wert des Analogeingangs 1 zu erhalten. Die Variable *ain1* bekommt das Empfangsbyte zugewiesen. Diese Variable ist mit dem *meter* - Element mit seinem Bereich 0 bis 5 verknüpft, so dass der Balken bei Vollauschlag 5 Volt am Eingang 1 signalisiert.

Im Zusammenhang mit den Zahlenumwandlungen aus dem ersten Kapitel erfolgt im nachstehenden Listing auch noch die Darstellung der acht Digitaleingänge als Bitmuster oder Binärzahl. Der Timer sorgt für ein langsames Lauflicht an den Digitalausgängen, indem ein Bit entsprechend verschoben wird. Die Abfrage der Eingänge erfolgt ebenfalls in dieser Timer-Routine, wobei die beiden Abfragen in Unterprogrammen ausgelagert sind. Die Umwandlungsroutine macht sich die Unterscheidung zwischen Groß- und Kleinschreibung bei Variablennamen zunutze.

memclear

Baudrate 19200

Meter ain1,0,5

Textbox ain1

**html | A1/Volt
|**

Meter ain2,0,5

```
Textbox ain2
html | A2/Volt<br>|
Meter dez,0,255
Textbox B$
html | DIN<br>|
serialbranch [rx]
Timer 500, [tm]
bit = 1
wait
[tm]
b$="……."
Gosub [dout]
bit = bit << 1
If bit > 128 then bit = 1
Gosub [ain]
Gosub [din]
wait
[rx]
serialinput r
ain1=asc(r)
Return
[ain]
SerialPrint chr(60)
ain1=serial.read.int() / 51
Serialprint chr(58)
ain2=serial.read.int() / 51
Return
[din]
SerialPrint chr(211)
dez=serial.read.int()
[dez2bin]
Dez = dez
For i = 0 to 7
mid(b$,8-i,1)=chr(48 + Dez % 2)
Dez = Dez>>1
next Bit
```

B$ = b$

return

[dout]

SerialPrint chr(81)

SerialPrint chr(bit)

Return

4.4.4 *ESPBASIC als HTTP-Server*

In Verbindung mit der *msg* -Methode von ESPBASIC kann das alte Interface jetzt auch von jedem Browser aus angesprochen werden. Die Steuerung erfolgt, wie weiter oben erläutert und bei der Excel-Steuerung schon angewandt, über die URL und somit über HTTP. Da Port 80 auch von BASIC selber benutzt wird, muss in der URL das Schlüsselwort *msg?* vorhanden sein, damit ESPBASIC die danach folgenden Parameter weiter reicht. Falls der ESP vom Router die IP 192.168.178.40 zugewiesen bekam, so kann ein digitales Muster 01010101 alias 85 vom Browser aus mit ***http://192.168.178.40/msg?cmd=dout&val=85*** erfolgen.

Voraussetzung ist, dass ein entsprechendes ESPBASIC-Programm läuft, welches diese Zeile entsprechend auswertet. Mit *msgbranch* und *msgget* entsteht ein Listing, welches es erlaubt mittels ESPBASIC das *CompuLab* -Interface über HTTP zu verwenden. Der HTTP-Zugang ist üblicherweise über Port 80 frei zugänglich, da das normale Internet im Browser auch darüber läuft.

```
memclear
Io(po,13,1)
Print “CLAB OVER HTTP”
serial2begin 19200, 5, 4 ‘TX/RX
serial2branch [rx]
msgbranch [msgbr]
wait
[msgbr]
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “ok”
If cm == “LED1” then Io(po,15,1)
If cm == “LED0” then Io(po,15,0)
‘+++++COMPULAB+++++++++++++++++
if cm == “AIN1” then
 Serial2Print chr(60)
 MyReturnMsg = ””
```

```
end if
if cm == “AIN2” then
 Serial2Print chr(58)
 MyReturnMsg = ””
end if
if cm == “DIN” then
 Serial2Print chr(211)
 MyReturnMsg = ””
end if
if cm == “DOUT” then
 Serial2Print chr(81)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
‘+++++COMPULAB+++++++++++++++++
If MyReturnMsg <> ”” then msgreturn MyReturnMsg
wait
[rx]
serial2input z$
ret = asc(mid(z$,1,1))
Msgreturn ret
Return
```

Die Initialisierung schaltet die große, blaue LED am Witty Cloud an, um zu zeigen, dass dieses Programm läuft. Außerdem erscheint im Browser und der seriellen Schnittstelle ein Hinweis auf dieses laufende Programm. Da ESPBASIC sehr geschwätzig ist und sich diese internen Ausgaben über die „normale“ serielle Schnittstelle nicht abschalten lassen, kommt es bei diesem Verfahren bei Verwendung der üblichen TX/RX-Leitungen zu unerwünschten Ergebnissen. Aus diesem Grund erfolgt die HTTP-Steuerung hier über eine zweite serielle Schnittstelle an den Leitungen GPIO05(D1) und GPIO04(D2). Anstatt TX/RX – am Witty Cloud sind das die beiden nächsten Anschlüsse. Mit *serial2begin 19200, 5, 4* wird diese Verbindung initialisiert. Der *msgbranch [rx]* dient dazu seriell ankommende Bytes entgegen zu nehmen. Die beiden Variablen *cm* und *va* erhalten über *msgget* die entsprechenden

Parameter als Zeichenkette bzw. Zahlenwert. Als kleiner Test, ohne das angeschlossene Interface, lässt sich die LED an Pin 15 (rot) mit *led1* und *led0* schalten, um die HTTP-Verbindung allgemein zu überprüfen. Im Browser ist das: *http://192.168.178.40/msg?cmd=led1*. Der Rückgabewert ist ein „ok“, wie in der Variablen *MyReturnMsg* voreingestellt.

Danach folgen die Abschnitte für die einzelnen Interface-Kommandos mit den entsprechenden Steuerbytes und möglichen Rückgaben über *msgreturn* für die angeschlossene Hardware. Eine Abfrage wartet nicht auf das Rückgabebyte sondern überlässt das dem Abschnitt [*rx*], der einlaufende Daten entgegen nimmt und als Zahlenwert dem Browser oder sonstigen Aufrufern weiter reicht.

4.4.5 Messen und Steuern in Excel über RS232 – ohne COM

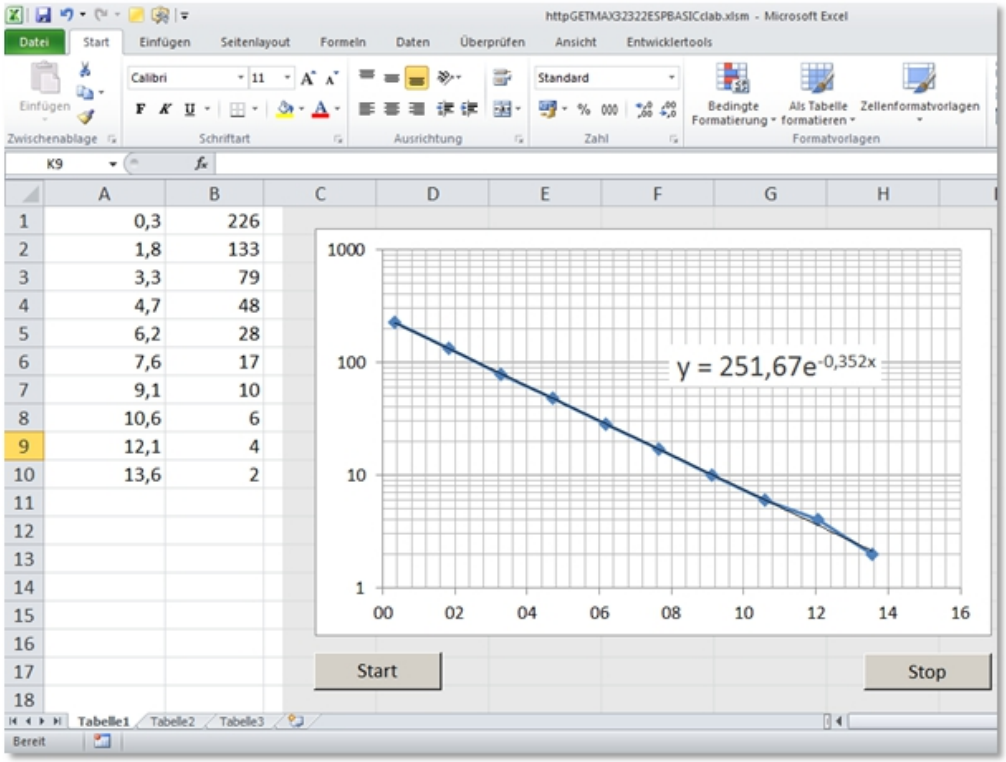


Abbildung 4-12: Messen in Excel mit RS232/WiFi-Adapter

Als Aufrufer soll nochmals Excel heran gezogen werden, um zu zeigen, dass auch ohne TCP/IP oder RSAPI.DLL aus [4] das Interface in Excel drahtlos über HTTP benutzt werden kann. Die weiter oben benutzte Widerstands-Kondensator-

Kombination soll mit entsprechenden Messdaten direkt in einem Diagramm erscheinen.

Mit einer Reihenschaltung von Widerstand (10k) und Kondensator (220µ) erfolgt eine Auf- und Entladung, wobei die Entladekurve direkt im Diagramm mit Trendlinie und logarithmischer Y-Achse erscheint. Zu beachten ist, dass die RX/TX Leitungen entsprechend dem ESPBASIC-Programm verbunden sind. Ein Digitalausgang dient als schaltbare Spannungsquelle zwischen 0 und 5 Volt (*DOUT*), als Eingang dient der Analogeingang A (*AIN1*). Bei dem Ergebnis der Zeitkonstanten - als negativer Kehrwert des Exponenten – ist das Abknicken der Kurve aufgrund der geringen Auflösung im unteren Spannungsbereich (ca. 20 mV pro Bit) der Kurve zu beachten, sowie den sich ergebenden Ungenauigkeiten der Zeitbestimmung am Empfangsort. Für genauere Messungen sollte die Zeit vom Messort mit übertragen werden, was hier nicht vorgesehen ist.

Um die gezeigte Darstellung und Messung zu erhalten sind einige wenige Zeilen in VBA/Excel erforderlich. Der Startknopf ruft die *sub Kondensator()* , der Stopp-Button springt nach *Sub Ende()* . Achtung, es folgt Visual Basic for Applications (VBA):

```
Function ESP ( ByVal Cmd$)

Set HTTP = CreateObject( "WinHttp.WinHttpRequest.5.1" )

    HTTP.Open "GET" , "http://192.168.178.40/msg?" + Cmd$

   HTTP.send

    ret = HTTP.ResponseText

    ESP = ret

    Set HTTP = Nothing

End Function

Sub Kondensator ()

  Range( "a:b" ).ClearContents

  DOUT 255

  Range( "b1" ) = "1"
```

```
  Range( "a1" ) = "Aufladen"
  While Range( "b1" ) < 250
   DoEvents
    Delay 200
    Range( "b1" ) = AIN1
  Wend
  DOUT 0 : t0 = Timer
  For zeile = 1 To 10
   Cells(zeile, 2 ) = AIN1
   Cells(zeile, 1 ) = Timer - t0
   calculate
  DoEvents
   Delay 1000
  Next zeile
  DOUT 0
End Sub
Sub DOUT ( ByVal wert)
 ESP ( "cmd=dout&val=" + Trim(Str$(wert)))
End Sub
Function AIN1 % ()
   AIN1 = Val(ESP( "cmd=ain1" ))
End Function
Sub Delay ( ByVal ms)
 t = Timer
 While Timer < (t + ms / 1000 )
  DoEvents
 Wend
End Sub
```

```
Sub Ende ()

  End

End Sub
```

Der Kondensator wird so lange aufgeladen, bis der Analogwandler den Wert 250 (4,9 Volt) meldet. Dann startet mit *DOUT 0* die Entladung über den 10k-Widerstand. Nach der Zeitinitialisierung folgt die Aufnahme und Darstellung von 10 Messwerten.

4.4.6 AUSBLICK

Nachdem beide Richtungen, Dank der ereignisorientierten Programmiermöglichkeit in ESPBASIC, sehr entspannt funktionieren, ist eine für das Gerät typische Gestaltung der Anzeige- und Steueroberfläche möglich. Wegen der geringen Maße der benötigten, zusätzlichen Bauteile ohne Stecker, könnte mit Hilfe eines ESP01S – der kleinste der Familie, jetzt aber mit mehr Speicher – eine Minimallösung im Gerät untergebracht werden.

Abbildung 4-13: Weitere Aufbaumöglichkeiten

ESP01 und MAX-Platine (ohne Stecker) könnten von innen an die Sub D-Buchse angeschlossen -, und die Buchse von außen gesperrt werden. Ein 3,3 Volt Regler liefert aus der internen 5 Volt Spannung die Speisung für ESP und MAX. Alternativ ist eine Stecker/MAX/ESP-Kombination als WiFi-Dongle für RS232-Geräte denkbar - mit Li-ion-Akku und USB-Lade-Elektronik.

Zum Zeitpunkt dieser Niederschrift ist dieser Weg mit einem ESP01S nicht möglich, da ESPBASIC und auch andere Sketche das Dateisystem (SPIFF) im ESP01S nicht lesen können. Wie im Netz [1] zu erfahren, gibt es inzwischen eine Lösung, die aber eine neue Übersetzung des ESPBASIC's erfordert, was bisher jedoch nicht erfolgt ist. Aus diesem Anlass und als weiterer Test entstand ein eigener Sketch in

C/C++ für die Arduino IDE, der nicht auf das Dateisystem angewiesen ist und das SIOS-Interface direkt unterstützt. Dieser Sketch erlaubt zusätzlich die Benutzung des Kommandos *TAIN* , um Messwert und Zeit unabhängig von der Übertragungszeit zu erhalten. Weiterhin kann durch die Implementierung von *Send()* und *Read()* auf unterster serieller Ebene mit RS232-Geräten über das *http://URL* Verfahren kommuniziert werden, ähnlich der Routinen *SendByte* und *ReadByte* der RSAPI.DLL aus [4].

Dieser Sketch für das *SIOS* -Interface sollte als Binärdatei unter der Adresse *hjberndt.de/soft/ESP01WebServSIOSrMsg1.ino.generic.bin* abrufbar sein und kann zum Beispiel mit der kostenlosen Android-App „ESP8266 Loader“ aus dem Playstore über einen Hostadapter direkt über USB vom Smartphone in den ESP übertragen werden. Das gilt natürlich auch für die Binärdateien von ESPBASIC auf Github.

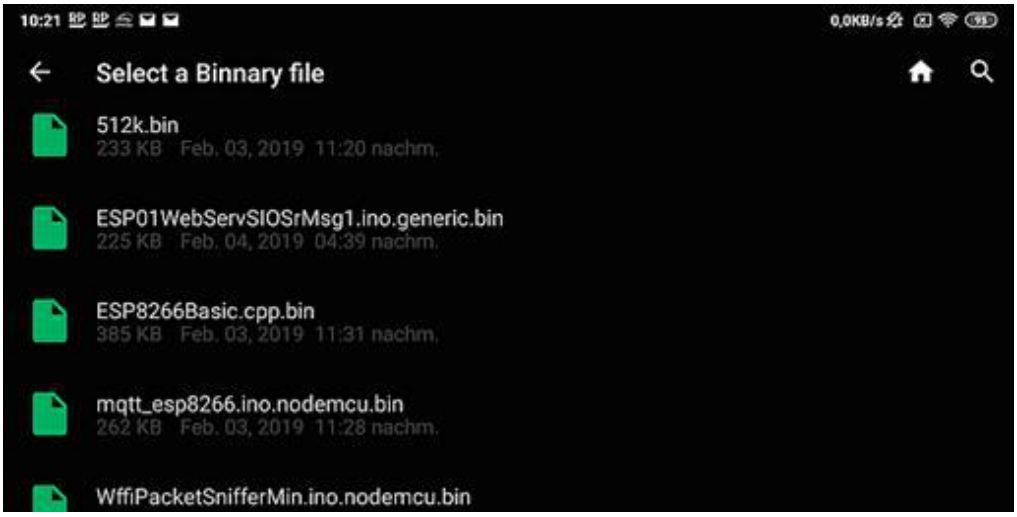


Abbildung 4-14: Binärdatei direkt übertragen mit der App „ESP8266 Loader“

Damit die Minimallösung auch mit dem jetzigen ESPBASIC funktioniert muss also ein ESP8266F, oder Ähnliche eingesetzt werden, der aber ohne Entwicklerboard auch nicht viel mehr Platz und Kapital erfordert, wie der kleine ESP01S.

Abbildung 4-15: D1 Mini ESP8266 und aufgelegtes MAX3232-Breakout (Suchbild)

Flach genug und preiswert für den Einbau ist das Gespann ESP8266 in der D1 mini-Variante mit dem kaum sichtbaren MAX3232-Board.

Im Anhang befinden sich die HTTP-Server-Listings für die RS232-Interfaces *CompuLab* , *Sios* und *CamFace* in einer erweiterten Version. Damit lassen sich z. B. Messzeitpunkt und Messwert zusammen abfragen.

4.5 Multimeter über WiFi/WLAN

Multimeter mit PC-Anschluss arbeiteten zunächst mit RS232-Schnittstellen, die aus Sicherheitsgründen über Optokoppler mit dem PC verbunden waren und noch sind. Einige Hersteller haben diese RS232-Adapter mit einem USB-Anschluss ergänzt, um sie an aktueller Hardware betreiben zu können. Die Vorteile solcher Messgeräte zur Messdatenerfassung liegen auf der Hand. Zu erwähnen seien nur die Messbereiche im mV oder µA-Breich, aber auch die Temperatur und Widerstandsmessung sind damit einfach zu erfassen. Ein Nachteil ist die geringe Messfolge, der aber im Zusammenhang mit ESPBASIC nicht ins Gewicht fällt.

Abbildung 4-16: Multimeter mit PC-Schnittstelle (RS232/USB)

Im folgenden Abschnitt soll die Möglichkeit überprüft werden, mit einem vorhandenen Multimeter mit PC-Schnittstelle drahtlos Messdaten über WiFi zu übermitteln. Es handelt sich hier um einen experimentellen Ansatz, der zunächst mit sicherheitstechnischen Mängeln behaftet ist, falls das ESP8266-Modul über ein USB-Netzteil betrieben wird und Messungen am Stromnetz erfolgen. Darum sei an dieser Stelle

ausdrücklich darauf hingewiesen, dass ein eventueller Nachbau auf eigene Gefahr erfolgt und der Autor keinerlei Haftung übernimmt.

Die Vorteile und Möglichkeiten eines solchen Ansatzes sind u.a.:

- Messungen über das Internet möglich
- kein Treiber erforderlich
- unabhängig vom Betriebssystem
- kabellos und zuverlässiger als Bluetooth
- mehrere ESP/ Geräte in z. B. Excel ansprechbar

Als Messgerät kommt ein älteres Voltcraft VC840 mit RS232-Adapter zum Einsatz. Aktuellere Geräte mit FS9721-Chip sind HoldPeak HP 90EPC, VC820/840, UNI-T 61B. Um ein Ergebnis in z. B. Excel zu erhalten sind Vorarbeiten auf Seiten der Hardware und der Software erforderlich.

4.5.1 Hardware des Multimeters

Um die seriellen Daten einem ESP8266 zuzuführen lassen sich verschiedene Wege beschreiten. Der dabei entstehende Aufwand, aber auch unerwartete Probleme können dann zum Verwerfen des gewählten Ansatzes führen.

1. Ansatz: Ungeöffnetes Gerät

   Das vorhandene RS232-Adapterkabel mit einem MAX3232-Adapter, wie im vorigen Abschnitt zu betreiben scheitert, da die Leitungen DTR und RTS nicht wie am PC entsprechende Spannungen führen, die zum Betrieb der Optokopplung nötig sind. Versuche, die Spannungen vom MAX-Baustein abzugreifen scheiterten aufgrund der hohen Belastung.

2. Ansatz: Geöffnetes Gerät

   Die Anschlüsse der Infrarotdiode sind beim VC840 mit Federkontakten versehen, die ein entsprechendes TX-Signal führen, allerdings in umgekehrter Polarität. Eine Invertierung mit OpAmp oder Transistor wäre möglich, aber aufwändig.

3. Ansatz: Geöffnetes Gerät

   Durch Internetrecherche gefundene Schaltpläne des VC840 zeigen ein FS9721-LP3 Chip mit TX-Anschluss an Pin 64, welcher sogar auf der Platine herausgeführt ist. Von dort erhält die IR-Diode über Widerstand und Transistor sein inverses Signal.

Ein vierter und letzter Ansatz bei ungeöffnetem Gerät schließt diesen Abschnitt weiter unten ab.

Der TX-Pin liefert ein 3 V-Signal mit positiven TTL-Pegel, welches direkt zum 3,3 V-ESP8266 Digitaleingang kompatibel ist. Zum Testen des Konzeps ist also keinerlei Zusatzhardware erforderlich.

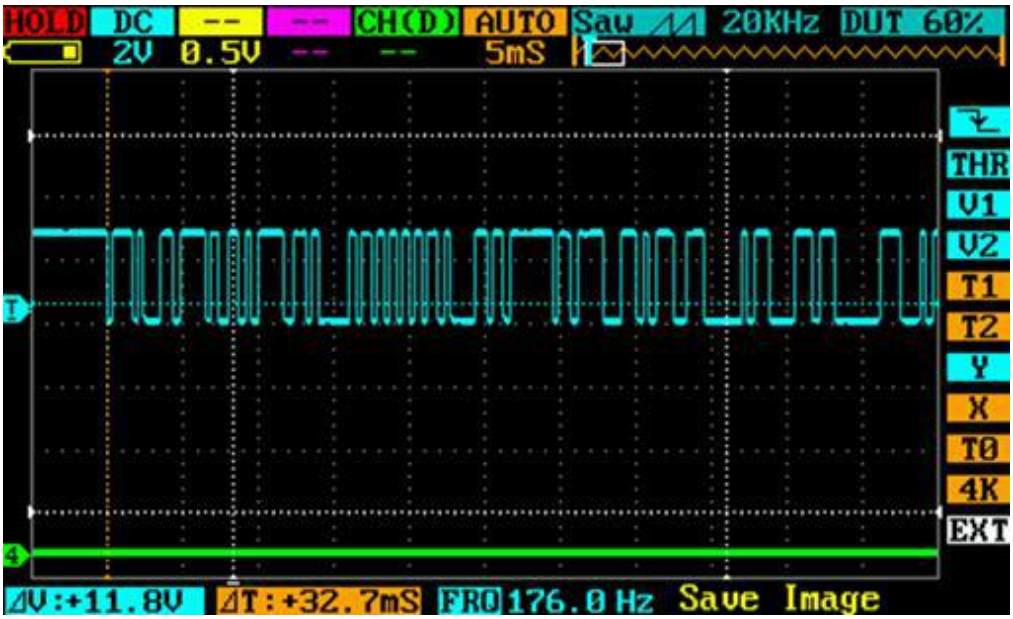


Abbildung 4-17: Logikpegel an TX-Pin 64 des Multimeterbausteins

4.5.2 Software zur Dekodierung

Das Gerät sendet mit der Übertragungsrate 2400 Baud und überträgt dabei keinen Klartext im ASCII-Format, sondern 14 Bytes, die Informationen darüber enthalten welche Segmente in der Anzeige angeschaltet sind. Eine Ziffer besteht aus sieben Segmenten, wodurch 7 Bit pro Ziffer erforderlich sind. Einheit, Messbereich und andere Dinge erfordern weitere Bits. Das Format ist genauer in einem Datenblatt von *tekpower.us* im Anhang 5.3.1 angegeben. Zunächst wird überprüft, ob überhaupt 14 Bytes empfangbar sind.

Das Testprogramm benutzt die *Serial2* -Aufrufe aus Abschnitt 0, um von den Leitungen RX/TX der Hauptkommunikation am ESP unabhängig zu sein. Ein *Serial2Branch* fängt die etwa alle 0,5 Sekunden eintreffenden Daten ein:

```
serial2begin 2400, D3, D2 'TX/RX
serial2branch [rx]
textbox n
x = -1
wait
[rx]
io(po,2,x)
x = not x
Serial2input z$
n = len(z$)
return
```

Der Empfangspin D2 am ESP D1 Mini erhält sein Signal von Pin 64 des Multimeterchips. Beide Geräte sind mit ihrer Masse (nicht der Minuspol der Batterie!) mit einander verbunden. Die kleine blaue LED sollte bei eingeschaltetem Multimeter etwa im Sekundentakt blinken, was bedeutet, dass ESPBASIC Daten erhält, da der *serial2branch* angesprungen wird. Die Bytes landen im Empfangspuffer *z$* , der bei korrekter Übertragung 14 Zeichen oder Bytes enthalten sollte. Die Längenabfrage liefert in *n* und damit in der *textbox* diese Zahl 14.

Die 14 Bytes sind in der höheren Tetrade, also in den Bits 7 bis 4 durchnummeriert, die niedrigere Tetrade enthält die Informationen zur Anzeige. Mit der Anweisung *r = asc(mid(z$,2,1)) >> 4* sollte *r* den Wert 2 erhalten, da das zweite Byte um vier stellen nach rechts geschoben die Nummer des Bytes selber sein sollte.

Ist die linke Ziffer im Display eine „0“ und es liegt kein negativer Messwert vor, so ist das 2. Byte eine 23, da 0010 0111 übertragen wird. Die linke Tetrade ist die 2 für das zweite Byte und die drei Bits rechts stehen für die Segmente *abc* der Anzeige, die bei einer „0“ alle gesetzt sind. Bit 3 zeigt mit der 0, dass das Vorzeichen nicht negativ ist. Der Rest der

Segmente der ersten Ziffer im Display ist in der rechten Tetrade von Byte 3 gespeichert, entsprechend der Segmente *defg*. Die Segmentbezeichnung mit Kleinbuschstaben weicht von den sonst üblichen 7-Segment-Angaben ab. An der Stelle des Vorzeichens (Bit 3) steht in den drei folgenden Bytes entsprechend der Dezimalpunkt.

Tabelle zur Segmentanzeige des Multimeters (vgl. Anhang 5.3.1):

a	**b**	**c**	**d**	**e**	**f**	**g**		**Dez**	**Hex**
0	0	0	0	1	0	1	1	5	05
1	0	1	1	0	1	1	2	91	5B
0	0	1	1	1	1	1	3	31	1F
0	1	0	0	1	1	1	4	39	27
0	1	1	1	1	1	0	5	62	3E
1	1	1	1	1	1	0	6	126	7E
0	0	1	0	1	0	1	7	21	15
1	1	1	1	1	1	1	8	127	7F
0	1	1	1	1	1	1	9	63	3F
1	1	1	1	1	0	1	0	125	7D
1	1	0	1	0	0	0	L	104	68
0	0	0	0	0	0	0		0	00

Damit kann nun testweise eine Ziffer dekodiert werden.

```
memclear
Textbox l
Textbox cnt
Textbox d
Textbox x
cnt = 0
‘Print “VC840”
Serial2begin 2400, D3, D2 ‘TX/RX
```

```
Serial2branch [rx]
Timer 2000,[tm]
wait
[tm]
b3 = asc(mid(z$,2,1))
b4 = asc(mid(z$,3,1))
b3 = b3 && 7
b4 = b4 && 15
x =  b4 + (b3 << 4)
d = "?"
if x =   5 then d = "1"
if x =  91 then d = "2"
if x =  31 then d = "3"
if x =  39 then d = "4"
if x =  62 then d = "5"
if x = 126 then d = "6"
if x =  21 then d = "7"
if x = 127 then d = "8"
if x =  63 then d = "9"
if x = 125 then d = "0"
if x = 124 then d = "L"
if x =   0 then d = " "
Wait
[rx]
serial2input z$
l = len(z$)
cnt = cnt + 1
Return
```

In der Routine *[tm]* , die vom *Timer* alle zwei Sekunden aufgerufen wird, erfolgt die Dekodierung der Segmentinformationen. In *x* werden die Informationen aus der jeweils rechten Tetrade von Byte 2 und 3 durch Maskierung und Schiebung als Byte gesammelt, die dann mit der angegebenen Tabelle in eine entsprechende Ziffer dekodiert werden. Dies erfolgt mit einfachen Abfragen in dezimaler Form, unter berücksichtigung von Speicherplatz und Geschwindigkeit der Interpretersprache. Die Anzahl der

Datenpakete summiert sich in *cnt* , die Variable *d* enthält die dekodierte Ziffer. Die vollständige Dekodierung des angezeigten Messwertes ist nachfolgend weiter unten gelistet.

4.5.3 Multimeter über WiFi in Excel

Mit den Möglichkeiten des ESPBASIC-HTTP-Servers (vgl. 2.6) ist es möglich mit dem ESP8266 zu kommunizieren. Ein kurzes Beispiel in Excel und VBA soll zeigen, wie Messdaten direkt im Tabellenblatt landen. Die Messrate liegt bei maximal etwa 5 Sekunden. Das Diagramm zeigt Werte im mV-Bereich bei offenen Anschlüssen des Multimeters.

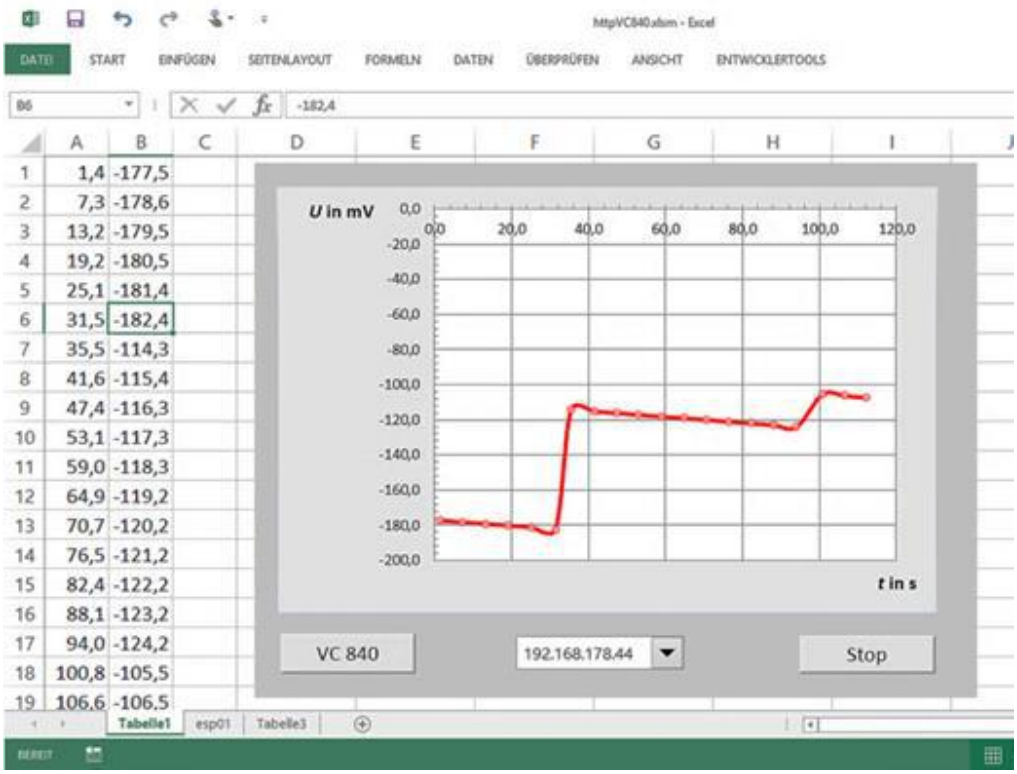


Abbildung 4-18: Multimeter online in Excel

Der als *default.bas* abgespeicherte http-Server in ESPBASIC startet nach 30 Sekunden automatisch, falls das in den Settings so eingestellt ist. Bei eingeschaltetem Multimeter und entsprechender Verbindung blinkt die blaue LED am ESP langsam und signalisiert damit eingehende Daten.

Excel ruft z. B. über 192.168.178.44 den Server im ESP8266, der die Routine *[mach]* aufruft und darüber die Dekodierung *[tm]* der 14 Bytes im Puffer aufruft und sie mit *msgreturn* per http-Protokoll an Excel weiter reicht.

```
memclear
y = -1
msgbranch [mac h]
Serial2begin 2400, D3, D2 'TX/RX
Serial2branch [rx]
wait
[mach]
gosub [tm]
msgreturn dis
wait
[tm]
dis = "+"
For xx = 2 to 8 step 2
b3 = asc(mid(z$,xx,1))
b4 = asc(mid(z$,xx+1,1))
b3 = b3 && 7
b4 = b4 && 15
x =  b4 + (b3 << 4)
d = "?"
if x = hextoint("05") then d = "1"
if x =  91 then d = "2"
if x =  31 then d = "3"
if x =  39 then d = "4"
if x =  62 then d = "5"
if x = 126 then d = "6"
if x =  21 then d = "7"
if x = 127 then d = "8"
if x =  63 then d = "9"
if x = 125 then d = "0"
if x = 104 then d = "L"
if x =   0 then d = " "
b3 = asc(mid(z$,xx,1)) && 8'+/-,
if xx = 2 then
 if b3 > 0 then dis = "-"
else
 if b3 > 0 then dis = dis&","
```

```
end if
dis = dis&d
next xx
return
[rx]
io(po,2,y)
y = not y
serial2input z$
return
```

VBA-Excel benutzt die Funktion *ESP()* , um die Daten über WiFi zu erhalten. Die IP ist hier im Excelblatt mit einem Dropdown wählbar (*k1,14*), kann natürlich auch direkt einprogrammiert sein. Mit der eigenen *delay* -Routine, die durch *doEvents* auch Endlosschleifen stoppen kann, läuft der Aufruf der Test *vc()* endlos.

```
Function ESP ( ByVal Cmd$)
Set http = CreateObject( "WinHttp.WinHttpRequest.5.1" )
    URL = Cells(Range( "k1" ), 14 )
    'http.Open "GET", "http://192.168.4.1/msg?" + Cmd$
    req = "http://" + URL + "/msg?" + Cmd$
    http.Open "GET" , req
   http.send
    ESP = http.ResponseText
    Set http = Nothing
End Function
Sub Delay ( ByVal ms)
 t = Timer
 DoEvents
 While Timer < (t + ms / 1000 )
  DoEvents
 Wend
```

```
End Sub

Sub Ende ()

 End

End Sub

Sub vc ()

While True

 t0 = Timer

 For i = 1 To 20

  Cells(i, 2 ).Select

  x = ESP( "" )

  Cells(i, 2 ) = CDbl (x)

  Cells(i, 1 ) = Timer - t0

 calculate

  Delay 5000

 Next i

Wend

End Sub
```

Wie zu Beginn erwähnt, ist dies bisher ein experimenteller Aufbau, der bei Messungen an Netzspannungen und Energieversorgung des ESP8266 mit einem USB-Netzteil nicht benutzt werden darf, um Lebensgefahr auszuschließen. Auch wenn die Anordnung mit einer Powerbank betrieben wird, bleibt die Gefahr für Dritte durch Unwissenheit bestehen. Abhilfe schafft die galvanische Trennung mit einem Optokoppler, wie beim Originalzubehör.

4.5.4 Adapter mit Optokoppler

Der mitgelieferte RS232-Adapter schiebt sich unter das Gerät und erhält die seriellen Daten optisch, so dass keinerlei elektrische Verbindung zwischen Messgerät und Empfänger

besteht. Der Datenstrom der IR-Diode am Messgerät ist mit Hilfe einer Digitalkamera erkennbar.

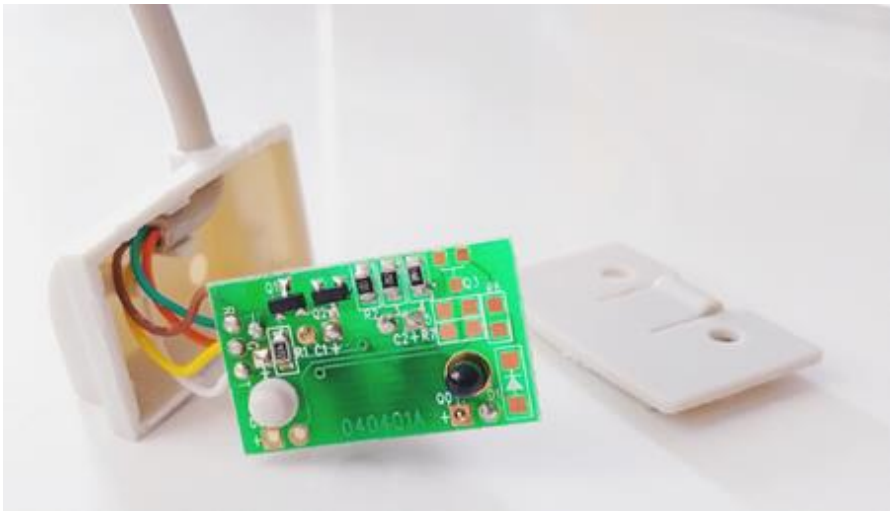


Abbildung 4-19: Geöffneter IR-Adapter mit Fototransistor (rechts)

Mittels Internetrecherche und einem scharfen Werkzeug ergibt sich folgendes Bild. Ein Fototransistor im Adapter nimmt die IR-Lichtsignale auf und zwei weitere Transistoren wandeln die Pegel entsprechend den RS232-Anforderungen.

Der RX-Teil mit weißem Anschluss ist nicht bestückt und auch die IR-Diode links ist nur eine mechanische Attrappe. Das gewünschte Signal steht am Punkt R_1/Q_0 zur Verfügung, wenn Q_1 abgetrennt ist. Da auch die negative Spannung des grünen Anschlusses nicht benötigt wird, reduziert sich der Adapter – ohne Zusatzkosten – auf folgende Schaltung:

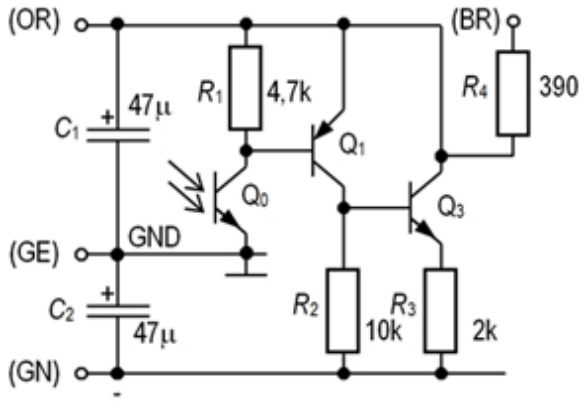


Abbildung 4-20: Schaltung des originalen IR-Adapters

Der unbenutzte, weiße Anschluss ist an der Platinen Bezeichnung „D1“ angeschlossen und überträgt nun das TTL-

Signal für den ESP8266 an Pin D2. Die Verbindung zwischen Q_0 und Q_1 ist unterbrochen.

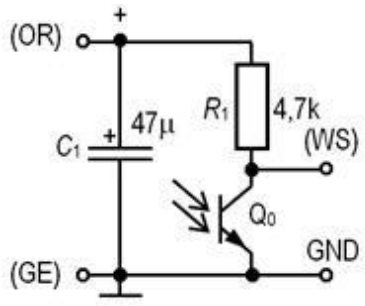


Abbildung 4-21: Modifizierter Adapter ohne Zusatzkosten

Adapterkabel	**ESP8266 D1mini**
Orange (OR)	Vcc (5 V)
Gelb (GE)	GND (Masse)
Weiß (WS)	D2 (Daten)

Die Verbindung des Adapters mit dem ESP8266 erfolgt mit den drei angegebenen Adern des aufgetrennten Kabels, alle anderen Adern werden nicht mehr benötigt.

Durch die Modifikation des IR-Adapters ist diese Anordnung wieder in einen sicherheitstechnisch unbedenklichen Zustand überführt worden. Das Board ESP8266 D1 Mini mit zugänglichem USB-Anschluss für die Spannungsversorgung ist dadurch wieder galvanisch vom Messgerät getrennt.

Aktuellere Geräte wandeln das RS232-Signal weiter in ein USB-Signal unter Verwendung eines CH9325 Chips. Dort steht das erforderliche TX-Signal in TTL-Pegel zur Verfügung, so dass ohne Mehraufwand ein WiFi-Anschluss realisierbar sein sollte.

Zur Untersuchung der Machbarkeit eignet sich ESPBASIC, wie hier gezeigt, ganz ausgezeichnet, auch wenn es ab und zu an Grenzen stößt. Der Entwicklung eines kommerziellen WiFi-Adapters für solche Messgeräte mit entsprechender

Hardware und Software in C stehen nur die Rechte des Autors im Weg.

4.6 16-Bit ADC Messungen – PGA – MUX – ADS1115

Der vorhandene analoge Eingang des ESP8266 in 10-Bit-Auflösung lässt sich mit einem ADS1115 erweitern. Damit könnten dann auch gegebenfalls die kleinen ESP01-Platinen analoge Messungen durchführen.

Abbildung 4-22: 4x Analog-Eingang mit 16 Bit und Verstärker: ADS1115

Der Baustein ist preiswert und benötigt lediglich zwei I2C-Leitungen um analoge Spannungen an insgesamt 4 Eingängen mit 16-Bit-Auflösung zu digitalisieren. Der Baustein lässt sich mit zwei verschiedenen Adressen betreiben, wodurch eine Erweiterung auf acht Analogeingänge mit dieser Auflösung problemlos möglich ist. Weiterhin verfügt der Baustein über einen programmierbaren Verstärker (PGA), um schwache Signale in verschiedenen Stufen anzuheben und einen Eingangsschalter (MUX), wodurch ein Betrieb als Differenzverstärker möglich ist. Die jeweilige Konfiguration steht in einem 16-Bit-Speicherplatz, dem sogenannten Konfigurationsregister.

4.6.1 Register und Konfiguration

Das Konfigurationsregister legt die Betriebsart des ADS1115 fest und hat folgenden Inhalt:

OS	**MUX**			**PGA**			**C/S**	**SPS**			**CM**	**CP**	**LC**	**CMODE**	
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	4	3	2	1	0
1	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	0	0	0	1	1
C				**1**				8				3			

Die wesentlichen Einstellungen finden in den Bits 9 bis 14 statt, da dort der Verstärkungsfaktor und die Eingangsverschaltung festgelegt werden. Mit der dargestellten Konfiguration $C183_{HEX}$ misst Eingang A0 gegen Masse mit 2/3 Verstärkung bei einer Wandler-Rate von 128 Samples pro Sekunde (SPS). Es werden Einzelmessungen (C/S) durchgeführt. Im Zusammenhang üblicher Anwendungen sind die grau hinterlegten Bits ohne Bedeutung. Mit jeweils 3-Bit-Feldern und damit 8 Möglichkeiten lassen sich die Einstellungen des Multiplexers, des Verstärkers und der Wandlungsrate einstellen. Die Wandlungsrate kann dabei die folgenden Werte annehmen, wobei 128 voreingestellt ist:

SPS/BIT	**7 6 5**
8	0 0 0
16	0 0 1
32	0 1 0
64	0 1 1
128	1 0 0
250	1 0 1
475	1 1 0

4.6.2 EINGANGSCHALTER UND PGA

Die Verstärkung des PGA (Programmable Gain Amplifier) hat laut TI-Datenblatt folgende Stufen, die in den Bits 5 bis 7 festgelegt sind. Die höchste Verstärkung liefert demnach 7,8 µV pro Bit! Damit lassen sich auch sehr schwache Signale direkt erfassen.

GAIN	**Auflösung pro Bit**	**11 10 9**
2/3	0,187500 mV	0 0 0
1	0,125000 mv	0 0 1
2	0,062500 mV	0 1 0
4	0,031200 mV	0 1 1
8	0,015620 mV	1 0 0
16	0,007812 mV	1 0 1
16	0,007812 mV	1 1 0
16	0,007812 mV	1 1 1

Vor dem PGA können die vier Eingänge und die Masse per Software umgeschaltet werden. Dieses Multiplexing zeigt TI in einer Skizze, die dieses Verhalten mit Schaltern verdeutlichen soll.

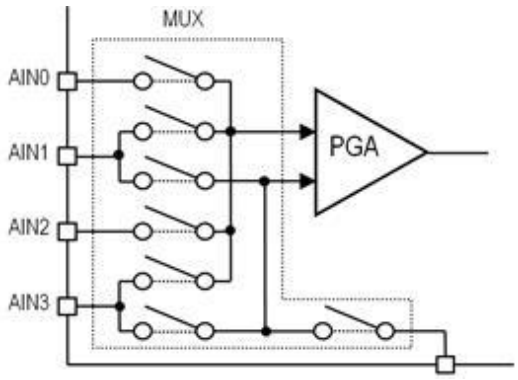


Abbildung 4-23: Eingangs-Umschaltung ADS1115

MUX
Bit 14 13 12

Bits	*PGA +*	*PGA -*
000	AIN0	AIN1
001	AIN0	AIN3
010	AIN1	AIN3
011	AIN2	AIN3
100	AIN0	GND
101	AIN1	GND
110	AIN2	GND
111	AIN3	GND

Wenn die Wandlerrate (SPS) konstant bleibt, ändern sich nur die Bits im MSB (höherwertiges Byte: Bit 15-8), wobei bei hexadezimaler Konfiguration das LSB (niederwertige Byte: Bit 7-0) gleich bleibt.

Um von ESPBASIC aus dieses Register zu verändern, reichen folgende Zeilen aus, wenn der Baustein mit den Leitungen SDA an GPIO0 und SCL an GPIO2 angeschlossen ist:

```
i2c.setup(0,2)
‘Konfigurieren
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(“c1”))‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
```

oder anders

```
i2c.setup(0,2)
config = hextoint(“c183”) ‘a0gnd
adr = hextoint(“48”) ‘ADS1115
hi = config / 256
lo = config and 255
i2c.begin(adr)
i2c.write(1) ‘ADS CONFIG
i2c.write(hi)
i2c.write(lo)
i2c.end()
```

Das Doppelbyte wird in Highbyte (MSB) und Lowbyte (LSB) geteilt, danach das Gerät adressiert, signalisiert, dass Konfigurationsdaten folgen und schließlich die beiden Bytes in der richtigen Reihenfolge gesendet.

4.6.3 EINKANALMESSUNG

Am Anfang steht eine Einkanalmessung mit der oben gezeigten Konfiguration: Messen von A0 gegen Masse bei 2/3 Verstärkung. Ein Bit entspricht dann 0,187500 mV und ein Faktor Volt-pro-Bit könnte den Wert 0,0001875 bekommen. Bei 16 Bit wäre das ein Messbereich von 12,288 Volt. Da der AD1115 auch vorzeichenbehaftete Spannungen liefern kann, halbiert sich der Bereich auf -/+ 6,144 Volt. Diese Angabe kann verwirrend erscheinen und zu der Annahme führen, dass auch Wechselspannung gemessen werden kann. Das ist nicht der Fall und führt zur Zerstörung. Der Hersteller betont, dass an den Eingängen A0 bis A4 die Maximalspannung Vcc+0,3 Volt, und die Minimalspannung GND-0,3 Volt , gegen Masse gemessen, betragen darf. Bei Unter- bzw. Überschreitung

erleidet der Baustein irreversible Schäden, was im Netzforen nachzulesen ist. Die negativen Spannungen liefert der ADS1115 bei Differenzmessungen, wie weiter unten beschrieben.

Im Browser erscheint ein Ergebnis mit einer besonders hohen V_{cc} -Pin-Angabe des Witty Cloud Boards.

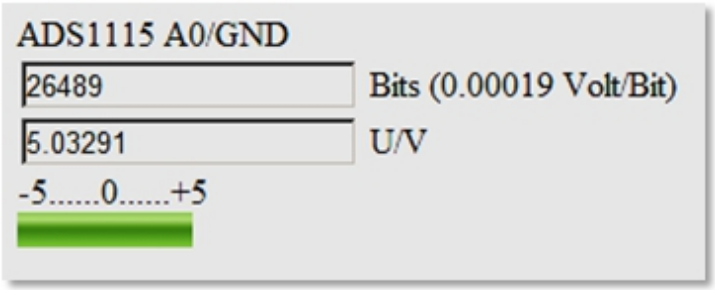


Abbildung 4-24: Einkanalmessung mit 0,19 mV/Bit

Listing zur Einkanalmessung:

```
print “ADS1115 A0/GND<br>”
i2c.setup(0,2)
vpb = 0.0001875’ Gain 0.67
textbox bits
wprint ” Bits (“&vpb&” Volt/Bit)<br>”
textbox u
wprint ” U/V<br>-5……0……+5<br>”
meter u,-5,5
timer 200 ,[messen]
wait
[messen]
‘Konfigurieren
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(“c1”))‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
‘Messwert abrufen
i2c.begin(hextoint(“48”))‘adr
i2c.write(0) ‘CONVERT
i2c.end()
i2c.requestfrom(hextoint(“48”),2)
msb = i2c.read()
lsb = i2c.read()
‘Spannung berechnen mit Vorzeichen
bits = msb * 256 + lsb
if bits > 32767 then bits = bits - 65536
u = bits * vpb
wait
```

Das Modul belegt die I2C-Leitungen GPIO 0 und GPIO 2 des ESP, die Darstellung erfolgt mit Web-GUI-Elementen wie *textbox* und *meter* . Der Messbereich der analogen Balkenanzeige ist auf +/- 5 Volt eingestellt, wobei der Nullpunkt in der Mitte bei halbem Ausschlag liegt. In den 200 ms-Intervallen erfolgt der Aufruf der Messroutine. Nach der

Konfiguration folgt der Aufruf der Messwerte über zwei Bytes, die entsprechend in eine Spannung umgerechnet und diese dann angezeigt werden.

4.6.4 MEHRKANALMESSUNG

Sollen mehrere Kanäle erfasst werden, so ist es erforderlich die Konfiguration vor jeder Kanalumschaltung zu ändern.

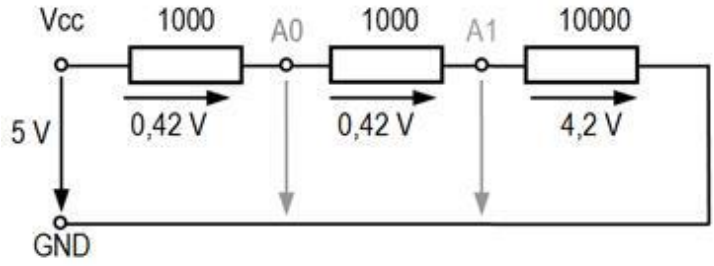


Abbildung 4-25: Aufbau zur Zweikanalmessung

Das MSB enthält die MUX-Bits und erfordert entsprechende Änderungen. Eingang A0 gegen Masse entspricht der Einkanalmessung „C1“, bei Eingang A1 gegen Masse ergibt die Tabelle den um eins erhöhten Wert „D1“. Das Listing der Einkanalmessung wird so angepasst, dass Konfiguration und Konversion in Unterprogrammen ausgelagert werden, um sie öfters aufzurufen. Die Kanäle werden in der Variablen *MSB* als Zeichenkette voreingestellt. Hier fällt die Unterscheidung von Groß- und Kleinschreibung in ESPBASIC auf, da *msb* im selben Listing auch auftaucht, aber anderen Inhalt hat. Die Benutzeroberfläche wird überwiegend kopiert und taucht nun doppelt auf. Ein Gesamt-Listing, das bei 12 Punkt Schriftgröße gerade noch so eben auf eine A4-Seite passt, folgt dem Ergebnis.

Die Ausgabe des Messergebnisses erzeugt das folgende Listing für einen Spannungsteiler an 5 Volt mit den Widerständen 1k - 1k - 10k in Reihe gegen Masse und A0 zwischen den beiden 1k Widerständen und A1 an 10k. Unbenutzte Eingänge liegen dabei an Vcc, wie vom Hersteller vorgeschlagen.

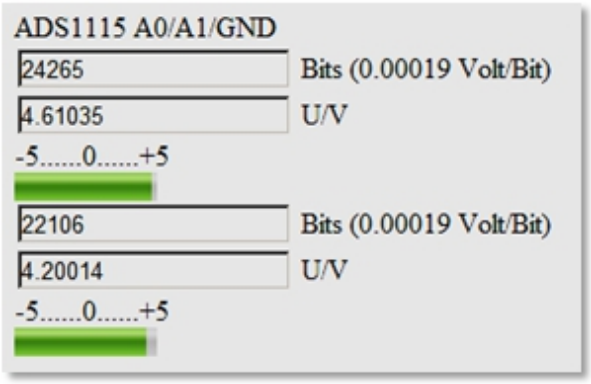


Abbildung 4-26: Ergebnis der Zweikanalmessung

**print “ADS1115 A0/A1/GND
”**

i2c.setup(0,2)

vpb = 0.0001875’ Gain 0.67

textbox bits0

**wprint ” Bits (“&vpb&” Volt/Bit)
”**

textbox u0

**html ” U/V
-5……0……+5
”**

meter u0,-5,5

**wprint “
”**

textbox bits1

**wprint ” Bits (“&vpb&” Volt/Bit)
”**

textbox u1

**html ” U/V
-5……0……+5
”**

meter u1,-5,5

timer 500 ,[messen]

wait

[messen]

MSB = “C1“‘KANAL A0gnd

gosub [config]

gosub [convert]

u0 = u’Messwert abrufen

bits0 = bits

MSB = “D1“‘KANAL A1gnd

gosub [config]

gosub [convert]

u1 = u’Messwert abrufen

bits1 = bits

```
wait
[config]
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
delay 10
return
[convert]
i2c.begin(hextoint(“48”))‘adr
i2c.write(0) ‘CONVERT
i2c.end()
i2c.requestfrom(hextoint(“48”),2)
msb = i2c.read()
lsb = i2c.read()
‘Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
return
```

Das Listing ist so aufgebaut, dass andere Kanalkombinationen ohne hohen Aufwand möglich sind. Das MSB für Kanal A2 bzw. A3 wäre „E1“ bzw. „F1“, laut Konfigurationsbits.

4.6.5 RC-SPANNUNGEN

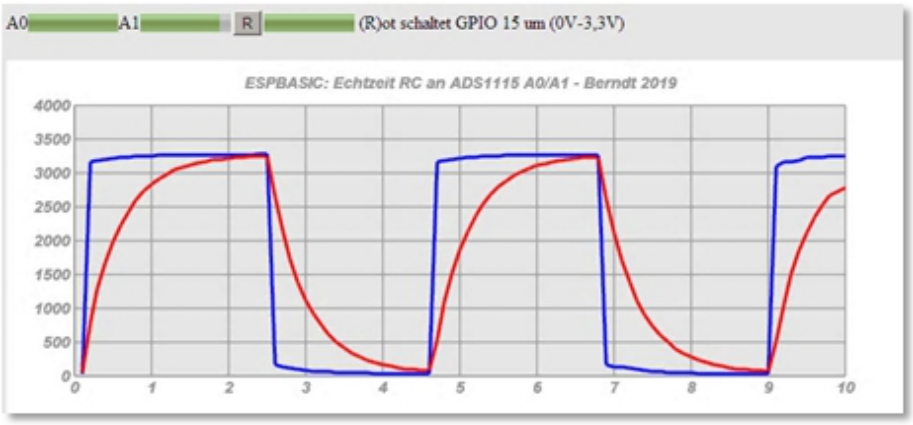


Abbildung 4-27: Auf- und Entladen mit JavaScript-Grafik

Ein Widerstand und ein Kondensator sind in Reihe geschaltet. Mit Hilfe eines Digitalausgangs (GPIO15) wechselt die Eingangsspannung zwischen 0 und 3,3 Volt als Rechteckspannung. Die Spannungen am Kondensator und am Eingang werden mit zwei Kanälen gegen Masse gemessen. Der Kondensator *C* soll wiederholt aufgeladen - und danach entladen werden. Eine grafische Darstellung erfolgt mit Hilfe von JavaScript in Echtzeit. Die Schaltung hat den folgenden Aufbau.

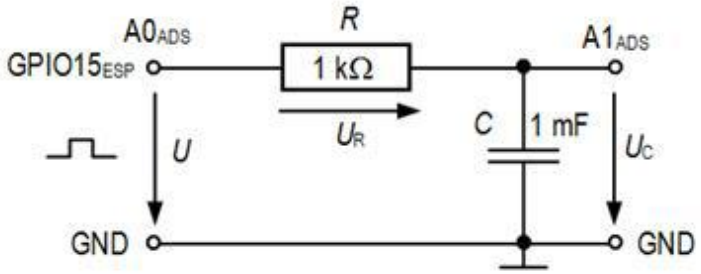


Abbildung 4-28: Aufbau zur RC Auf- und Entladung

Die Bauteile sind so dimensioniert, dass die Zeitkonstante aus Widerstand und Kapazität eine Sekunde beträgt, so dass praktisch spätestens nach 10 Sekunden ein Lade- und Entladezyklus absolviert ist.

Zu Beginn sei der Kondensator leer und $U_C = 0$, $U_R = 0$. Wechselt GPIO15 von 0 auf 3,3 Volt lädt sich der Kondensator über den Widerstand auf. Der zunächst hohe Ladestrom erzeugt an *R* eine proportionale Spannung, die wie der Strom

langsam abnimmt. Die Kondensatorspannung U_C steigt bis fast 3,3 Volt an. Nach fünf Zeitkonstanten ist der Kondensator praktisch voll. Zum Entladen des Kondensators wird GPIO15 auf 0 Volt geschaltet, was praktisch einer Verbindung zur Masse entspricht. Der Kondensator ist nun die Spannungsquelle im Stromkreis, die sich über den Widerstand entlädt. Während dieser fünf Sekunden sinkt die Kondensatorspannung U_C exponentiell, erreicht zur sogenannten Halbwertszeit die halbe Anfangsspannung und lässt den Entladestrom umgekehrt durch den Widerstand fließen. Das hat zur Folge, dass auch die am Widerstand anfallende Spannung in die andere Richtung zeigt, also negativ ist. Diese negative Spannung erhält man im obigen Aufbau, indem A0 und A1 jeweils gegen Masse (GND) gemessen wird und man anschließend die Differenz A0 – A1 bildet. Im nächsten Abschnitt wird dies mit dem ADS direkt gelöst.

Ein erster Test ohne Diagramm ist mit ESPBASIC schnell programmiert. Es ist im Prinzip eine Mehrkanalmessung und darum kann das Listing von dort übernommen werden. Hinzu kommt die Steuerung, die dafür sorgt, dass am Eingang der Schaltung ein langsames Rechteck erscheint. Durch den zweiten *Timer* in ESPBASIC schaltet GPIO15 im eingestellten Intervall (z. B. 5 Sekunden) um. Um nicht das gesamte Listing nochmals abzubilden folgen nur die Modifikationen. Der zweite *Timer* nennt sich *timercb* und steuert den Ausgang im Abschnitt *[steuern]*. Hier die Änderung:

Aus

...

meter u1,-5,5

timer 500 ,[messen]

wait

…

wird

meter u1,-5,5

timer 500 ,[messen]

timercb 5000,[steuern]

wait

Die Steuerroutine als Unterprogramm mit dem Sprungziel *[steuern]* kann ganz am Ende stehen:

```
[steuern]
io(po,15,x)
x = not x
return
```

Damit das Umschalten funktioniert, muss die Zeile *x = -1* am Anfang des Programms eingefügt werden. Die Diagrammvariante benutzt zum Steuern einen Taster und ist wegen der Übersicht als *Listing im Anhang 5.3.6* aufgeführt.

4.6.6 Differenzmessung – negative Spannungen

In der Technik ist es üblich Spannungen relativ zur Masse zu messen. Erfolgt eine Spannungsangabe an einem Messpunkt, so gilt diese Spannungsangabe bezogen auf Masse, was bei einigen Geräten auch das Gehäuse sein kann.

Eine besondere Eigenschaft des ADS1115 ist, dass Differenzmessungen zwischen den Eingängen möglich sind. Der Bezugspunkt ist dann nicht mehr die Masse (GND), sondern dieser wird frei festgelegt, ähnlich wie Batterie-Multimeter ohne Masseverbindung auch Spannungen „in“ der Schaltung anzeigen können und nicht nur bezogen auf Masse. Diese Wirkung kann auch bei der Mehrkanalmessung gegen Masse durch eigene Subtraktion erreicht werden, wobei allerdings zwei Konfigurationen und zwei Messungen pro Ergebnis nötig sind. Bei der Mehrkanalmessung weiter oben gilt die Spannung am mittleren Widerstand als positiv (Pfeilrichtung), wenn sie aus A0 – A1 berechnet wird. A0 und A1 messen gegen Masse. Rechnet man umgekehrt, also A1 – A0, so ergibt sich ein negativer Wert (entgegen der Pfeilrichtung). Noch besser lässt sich dieser Effekt bei der RC-Schaltung beobachten, wie dort beschrieben.

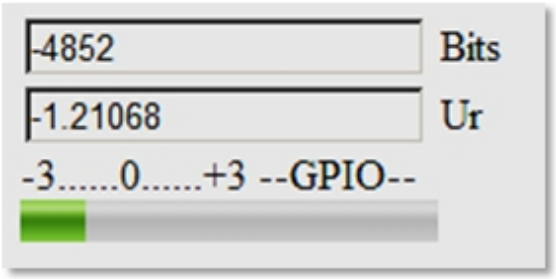


Abbildung 4-29: Negative Spannungsanzeige mit ADS1115

Zur Messung im Differenz-Betrieb kann das folgende Listing Verwendung finden. Eine karge Oberfläche zeigt den Digitalwert *Bits* der Differenzmessung zwischen A0 und A1. Der vorzeichenbehaftete Integer Wert wird für BASIC in der Zeile mit *bits-65536* errechnet, da BASIC so etwas nicht kennt. Darunter steht die sich berechnende Spannung U_R am Widerstand *R* mit dem Faktor *vpb* (Volt-Pro-Bit) mit sich änderndem Vorzeichen – also negativen Werten. Ein *meter* mit dem Bereich -3 V bis + 3 V zeigt diese Spannung quasi analog an. Das *meter* -Element daneben stellt den Zustand von GPIO15, also des Rechtecks dar. Da gerade entladen wird, ist dieser Balken „aus“. Man beachte, dass nur eine Messung erfolgt und sich die Konfiguration nicht ändert.

```
‘adsrcdiffa0a1.bas
memclear
i2c.setup(0,2)
vpb = 0.0001875
x = -1
textbox bits
html ” Bits<br>”
textbox u
html ” Ur<br>-3……0……+3 —GPIO—<br>”
meter u,-3.3,3.3
meter x,-1,0
timercb 5000,[steuern]
timer 250,[messen]
wait
[steuern]
io(po,15,x)
x = not x
```

```
return
[messen]
MSB = “81” ‘a0a1
gosub [config]
gosub [convert]
wait
[config]
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
delay 10
return
[convert]
i2c.begin(hextoint(“48”))‘adr
i2c.write(0) ‘CONVERT
i2c.end()
i2c.requestfrom(hextoint(“48”),2)
msb = i2c.read()
lsb = i2c.read()
‘Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
return
```

Wie weiter oben schon beschrieben, kann diese Differenzmessung auch aus zwei Einzelmessungen gewonnen werden. Das grafische Beispiel der Mehrkanalmessung in angepasster Form zeigt das entsprechende Diagramm.

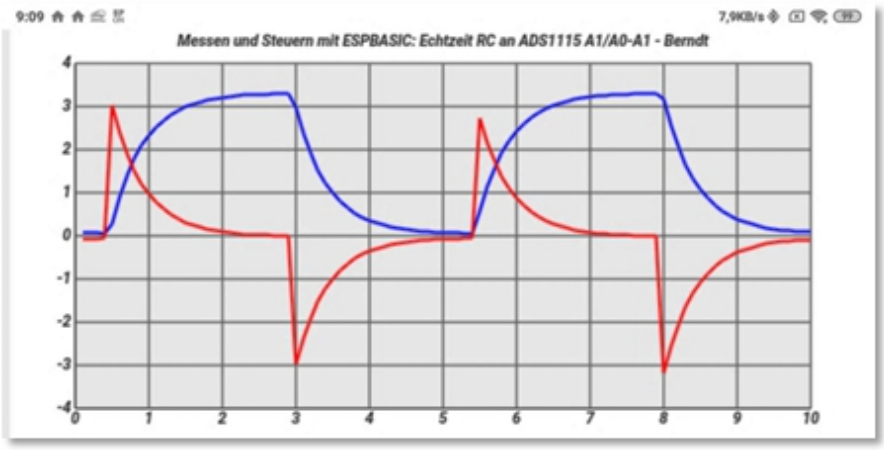


Abbildung 4-30: Strom- und Spannungsverlauf

Die erfolgten Änderungen sind ein veränderter *meter* -Bereich für die Anzeige der Differenzspannung. JavaScript bekommt den Wert vom *meter* -Element, muss also negative Werte zulassen. Die Y-Achse des Diagramms hat nun den Bereich -4000 bis +4000 mit 1000er Intervall. Die Spannung ist in mV angegeben. Das erzeugende *Listing ist im Anhang 5.3.8* aufgeführt und gestattet die Steuerung der Messung über einen Button, der zwischen Auf- und Entladen umschaltet, einem Button, der einen Timer mit fünf Sekunden-Intervall startet und damit zwischen Auf- und Entladung wechselt, sowie eine Interrupt gesteuerte Taster Abfrage am ESP Witty Cloud, um die Messung von dort aus zu steuern.

4.6.7 Eingangsverstärkung

Bei Betrachtung der Tabelle für die Bits 5 bis 7 im Konfiguratiosregister fällt die 16fache Verstärkung ins Auge. Damit ist eine Auflösung von 7,8 Mikrovolt pro Bit möglich. Der Messbereich berechnet sich dann mit 15 Bit zu 32768 mal 7,8 µV, also +/- 256 mV!

Bei dieser Verstärkung können auch sehr kleine Spannungsgeber noch ohne zusätzliche Verstärker für Messzwecke benutzt werden. Ein Sensor mit eher geringer Spannungsabgabe, wenn keine weitere Elektronik bemüht wird, ist das Thermoelement.

4.6.8 Thermoelement: Mikrovolt pro Kelvin

Ein Thermoelement besteht im einfachsten Fall aus der Verbindung zweier unterschiedlicher Metalle. Wird diese Verbindung einer Temperatur ausgesetzt, so entsteht eine geringe Spannung. Der Seebeck-Effekt beschreibt dieses -, der Peltier-Effekt das umgekehrte Verhalten.

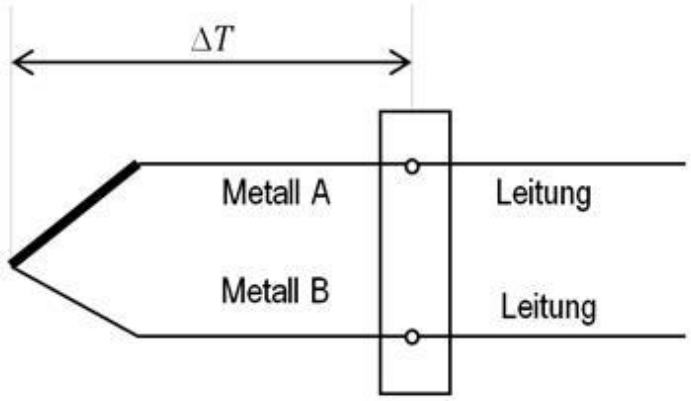


Abbildung 4-31: Thermoelement schematisch

Verdrillt oder verschweißt man Kupfer- und Konstantan Draht, so entsteht eine Spannung von 42 µV/K. Maßgebend für die entstehende Spannung ist die Temperaturdifferenz D*T* zwischen den beiden Verbindungspunkten. Bei der Verwendung von schmelzendem Eis am Anschlusspunkt, könnte die entsprechende Temperatur an der Messspitze direkt erfasst werden. Ist die Umgebungstemperatur konstant und bekannt, kann das Programm sie berücksichtigen und einrechnen.

Ab und zu findet man Temperaturadapter für Multimeter, die mithilfe einer Verstärkerschaltung die der Temperatur proportionale Spannung im mV-Bereich anzeigen, wobei dann 20 mV der Temperatur 20 °C entsprechen.

Mit dem ADS1115 kann eine solche Messung direkt erfolgen. Da die Empfindlichkeit relativ hoch ist, kann es sinnvoll sein sich langsam und vorsichtig an die höchste Verstärkungsstufe heran zu testen. Aus diesem Grund sind im Listing mehrere auskommentierte Konfigurationen aufgeführt, die in den Vorversuchen zur Anwendung kamen. Die blinkende, rote LED an Pin 15 dient ebenfalls nur dazu zu zeigen, dass eine Test-Messung läuft. In der letzten Teststufe gilt die Konfiguration

config = hextoint(“cb83”)‘a0gndx16

mit 16facher Verstärkung für Eingang A0 gegen Masse. Auch der Messbereich, bzw. die Messauflösung ist teileweise auskommentiert und berechnet schließlich mit 7,9 µV/Bit die Temperatur an der Messstelle, wobei hier die Umgebungstemperatur 18 °C an der Vergleichsstelle beträgt. Jetzt reagiert die Drahtverbindung auf jeden Hauch, da nun ja praktisch bei dieser Metallkombination eine Auflösung von 0,2 °C vorliegt.

u = hl * 7.8125’x8 uV

t = 18 + round(u/42*10)/10

Die Temperatur erscheint mit einer Nachkommastelle in der Anzeige. Das gesamte Listing hat den folgenden provisorischen Aufbau:

```
u = 0
x = -1
io(po,15,x)
html “7,9 uV pro Bit - 42uV/K<hr>”
textbox hl
html “Bit<br>”
textbox u
html “uV<br>”
textbox t
html ” C<br>”
meter t,0,50
timercb 5000,[steuern]
timer 1000 ,[messen]
wait
[steuern]
io(po,15,x)
x = not x
return
[messen]
adr = 72
‘i2c.setup(0,2)
‘config = 53635’a1
```

```
‘config = 49539’a0
‘config = hextoint(“c183”)‘a0gnd
‘config = hextoint(“9383”)‘a0a3x1
‘config = hextoint(“9583”)‘a0a3x2
‘config = hextoint(“9783”)‘a0a3x4
‘config = hextoint(“c783”)‘a0gndx4
‘config = hextoint(“c983”)‘a0gndx8
config = hextoint(“cb83”)‘a0gndx16
h = config / 256
l = config and 255
i2c.begin(adr)
i2c.write(1) ‘ADS CONFIG
i2c.write(h)
i2c.write(l)
i2c.end()
delay 10
i2c.begin(adr)
i2c.write(0) ‘ADS CONVERT
i2c.end()
i2c.requestfrom(adr,2)
h = i2c.read()
l = i2c.read()
hl = h *256 + l
if hl>32767 then hl=hl-65536
u = hl * 0.0001875’x.6
u = hl * 0.0000312’x4
u = hl * 15.625’x8 uV
u = hl * 7.8125’x8 uV
t = 18 + round(u/42*10)/10
wait
```

4.6.9 WIDERSTANDSMESSUNG

Eine einfache Methode den Wert eines unbekannten Widerstandes zu ermitteln – ganz ohne Strommessung - erfolgt über die Gesetzmäßigkeit, dass sich Spannungen wie Widerstände verhalten, also an einem großen Widerstand eine hohe Spannung abfällt usw..

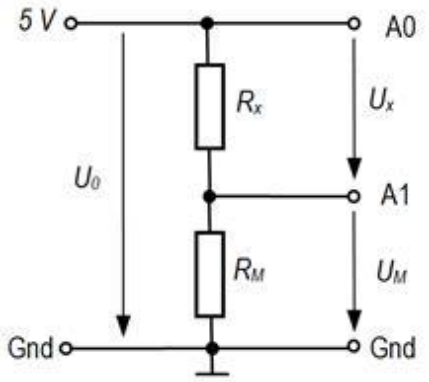


Abbildung 4-32: Aufbau Widerstandsmessung

Das Schaltbild zeigt eine Reihenschaltung an 5 Volt bestehend aus dem unbekannten Widerstand R_x und dem bekannten Widerstand R_m. Da sich in dieser Reihenschaltung die Widerstände wie die Spannungen verhalten, gilt ein Verhältnis, welches nach R_x aufgelöst ist:

$$\frac{R_x}{R_m} = \frac{U_x}{U_m}$$

$$R_x = R_m \cdot \frac{U_x}{U_m}$$

Die Spannung U_x ist in der Schaltung die Differenz zwischen A0 und A1 oder $U_0 - U_m$, die Spannung U_m wird über A1 direkt gemessen.

Hier: $R_M = 1000$ Ohm, $U_0 = 3{,}3$ V

Umgesetzt in ESPBASIC und dem ADS1115 führt folgendes Listing zum Ergebnis.

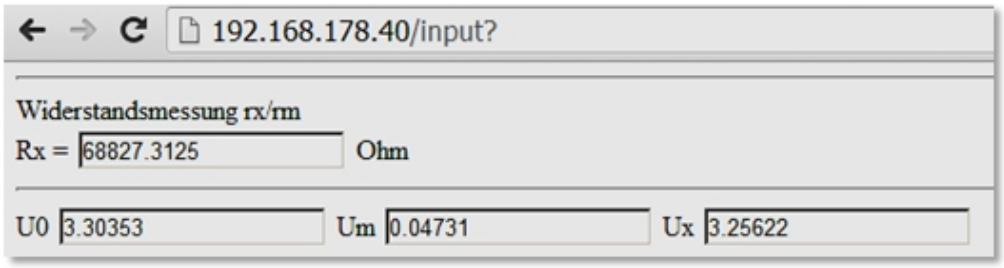


Abbildung 4-33: Ergebnis Widerstandsmessung für Blau-Grau-Orange-Gold - Widerstand

Nachdem die „Benutzeroberfläche“ gestaltet ist, sorgt *io(po,15,1)* dafür, dass GPIO 15 auf 3,3 Volt gelegt wird und so als U_0 dient. Die Messung wiederholt sich einmal pro Sekunde und berechnet in der Routine *[messen]* den Widerstandwert R_x aus den Spannungen an A0/A1. Zur Kontrolle zeigt die Oberfläche alle Spannungen an, was bei einer direkten Differenzmessung nicht möglich wäre.

```
print “Widerstandsmessung rx/rm <br>”
memclear
rm = 1000
vpb = 0.0001875
html | Rx = |
textbox rx
html | Ohm<hr>|
html “U0 “
textbox u0
html ” Um “
textbox um
html ” Ux “
textbox ux
io(po,15,1)
timer 1000,[messen]
wait
[messen]
MSB = “C1“‘KANAL A0gnd
gosub [config]
```

```
gosub [convert]
u0 = u ‘Messwert abrufen
MSB = “D1“‘KANAL A1gnd
gosub [config]
gosub [convert]
um = u ‘Messwert abrufen
ux = u0 - um
rx = rm * ux / um
wait
[config]
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
delay 2
return
[convert]
i2c.begin(hextoint(“48”))‘adr
i2c.write(0) ‘CONVERT
i2c.end()
i2c.requestfrom(hextoint(“48”),2)
msb = i2c.read()
lsb = i2c.read()
‘Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
return
```

4.6.10 Messung einer Induktivität - Resonanz

Mit Hilfe der einstellbaren Frequenz des PWM-Signals, soll die Möglichkeit untersucht werden, die Induktion einer Spule mit Eisenkern zu bestimmen. Die Gesetzmäßigkeiten der sinusförmigen Wechselspannung könnten trotz Rechteckfrequenz zu einem näherungsweise brauchbaren Ergebnis führen. Wie in Abschnitt 2 bereits angewendet, lässt sich mit *pwmfreq* das Signal eines PWM-Ausgangs in seiner Frequenz von 1 bis 8000 Hertz variieren. Durch eine Reihenschaltung von Spule und Kondensator entsteht ein Schwingkreis mit einer Spannungserhöhung bei der Resonanzfrequenz f_0. Mit Hilfe einer Diode erfolgt die Gleichrichtung des Wechselsignals hoher Spannung mit anschließender Glättung durch einen Kondensator. Die der Amplitude proportionale Gleichspannung steht dann als Messsignal zur Verfügung. Wegen der Spannungshöhe kann eine weitere Spannungsteilung mit zwei Widerständen erforderlich sein.

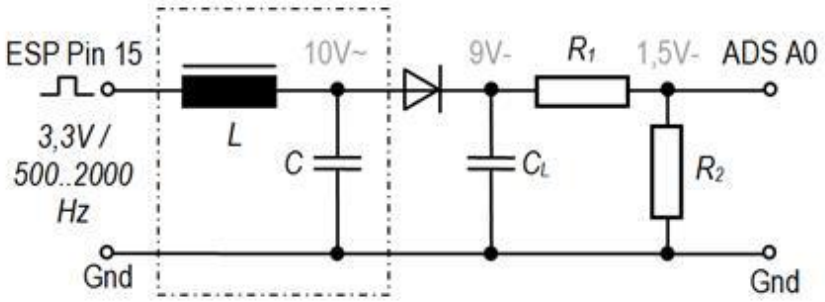


Abbildung 4-34: Induktivität und Resonanz

Als Messaufbau dient ein ESP8266 Modul Witty Cloud mit angeschlossenem ADS1115. Da der Analogeingang belegt ist, erfolgt die Messung mit dem externen A/D-Wandler an Eingang A0, entsprechend einer Einkanalmessung weiter oben. Der ESP liefert das Rechtecksignal unterschiedlicher Frequenz an Pin 15 mit der roten LED.

Als Messobjekt der Induktivität *L* liegt eine rechteckige Spule mit 1000 Windungen und Kern vor, deren Induktivität bestimmt werden soll. Als Kapazität steht ein Kondensator *C* mit 0,33 µF zur Verfügung. Der Messwandler mit Diode,

Ladekondensator und Spannungsteiler kann wie folgt dimensioniert sein: 1N4148, 10µ, 100k, 10k (v. l. n. r.). Kritisch ist hier die Messspannung am 10k-Widerstand, da der ADS1115 auf Überspannungen fatal reagiert! Sie sollte deutlich unter der Versorgungsspannung bleiben um Schäden zu vermeiden. Aus diesem Grund ist der Maximalwert hier mit nur ca. 1,5 Volt in der Mitte, um Reserven nach oben zu haben.

Mit Taschenoszilloskop und Multimeter erfolgen der Zusammenbau und die Überprüfung zunächst ohne ADS. Ein Oszillogramm zeigt die Spannung zwischen *L* und *C* gegen Masse (Blau) und die für den ADS gleichgerichtete und geglättete Spannungsteiler-Spannung in Gelb.

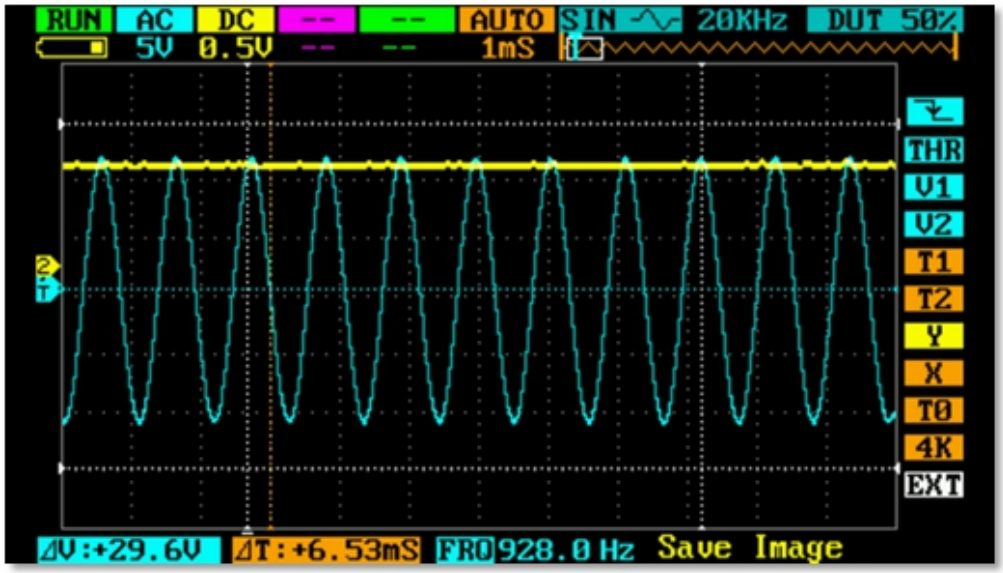


Abbildung 4-35: DSO-Quad zeigt Wechsel- und Gleichspannung

Messwertaufnahme und Auswertung sollen später direkt in Excel erfolgen, für erste Tests ist die Browserausgabe mit den Möglichkeiten von ESPBASIC vorzuziehen. Zwei *Textboxes* und ein *Slider* reichen zur Überprüfung. Eine Timer-Routine stellt die Frequenz ein und misst die Spannung am A0.

```
i2c.setup(0,2)
timer 500, [tm]
Textbox u
Slider f,2,8000
Textbox f
Io(pwo,15,512)
```

```
wait
[tm]
Pwmfreq f
Gosub [ain1]
‘f = f + 20
‘If f > 2000 then f = 500
wait
[ain1]
MSB = “C1”
gosub [config]
gosub [convert]
return
[config]
i2c.begin(72)
i2c.write(1)
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
return
[convert]
vpb = 0.0001875
i2c.begin(72)
i2c.write(0)
i2c.end()
i2c.requestfrom(72,2)
msb = i2c.read()
lsb = i2c.read()
bits = msb * 256 + lsb
if bits > 32767 then bits = bits - 65536
u = bits * vpb
return
```

Wer einen ESP8266 ohne verschalteten Analogeingang besitzt, oder den LDR des Witty Cloud ablötet, kann auf die I2C-Steuerung ganz verzichten, was in einem deutlich kürzeren Listing münden würde. Werden die beiden auskommentierten

Zeilen wieder aktiv, so läuft die Frequenz automatisch hoch und man hat die Hände frei, um eventuell einzugreifen.

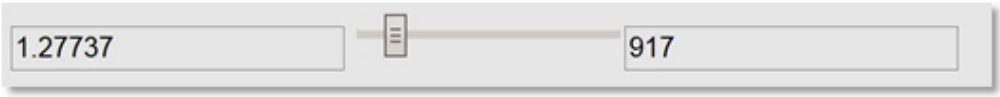


Abbildung 4-36: Manuelle Frequenzsuche

Wenn alles zur Zufriedenheit funktioniert, kann der ESPBASIC-Server für Excel zusammengestellt werden. Die LED schaltet nun im PWM-Modus *io(pwo,15,512)* mit einem Tastverhältnis von 0,5, also halbe Intensität oder symmetrisches Rechteck. Der Aufruf *freq* stellt die PWM-Frequenz ein. Ansonsten werden nur die Eingänge A0 und A1 des ADS1115 unterstützt.

```
memclear
Print “PWM & ADC over WiFi”
i2c.setup(0,2)
u = 1.2345678
msgbranch [br]
wait
[br]
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “88”
if cm == “AIN1” then
 Gosub [ain1]
 MyReturnMsg = str(u)
end if
if cm == “AIN2” then
 Gosub [ain2]
 MyReturnMsg = str(u)
end if
If cm == “LED1” then Io(pwo,15,512)
If cm == “LED0” then Io(pwo,15,0)
If cm == “FREQ” then pwmfreq va
if cm == “LDR” then
 MyReturnMsg = str(io(ai))
end if
```

```
msgreturn MyReturnMsg
wait
[ain1]
MSB = “C1”
gosub [config]
gosub [convert]
return
[ain2]
MSB = “D1”
gosub [config]
gosub [convert]
return
[config]
i2c.begin(72)
i2c.write(1)
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
return
[convert]
vpb = 0.0001875
i2c.begin(72)
i2c.write(0)
i2c.end()
i2c.requestfrom(72,2)
msb = i2c.read()
lsb = i2c.read()
bits = msb * 256 + lsb
if bits > 32767 then bits = bits - 65536
u = bits * vpb
return
```

Damit das auf der Gegenseite in VBA auch funktioniert, arbeiten dort folgende Routinen in einem eigenen Modul:

```
Function ESP ( ByVal Cmd$)

Set http = CreateObject( “WinHttp.WinHttpRequest.5.1” )
```

```
    http.Open "GET" , "http://192.168.178.40/msg?" + Cmd$
   http.send
    ESP = Replace(http.ResponseText, "." , "," )
    Set http = Nothing
End Function
```

Hier werden die Daten zwischen ESP und VBA ausgetauscht. Die analogen Spannungswerte kommen vom ADS direkt in der Einheit Volt, da die Umrechnung mit Verstärkung hier in VBA zu unpraktisch erscheint:

```
Function UIN ( ByVal nr) As Double ' U in Volt
  UIN = CDbl (ESP( "cmd=ain" + Trim(Str$(nr))))
End Function
```

Frequenzeinstellung und Delay sind kurze Subs:

```
Sub Frequenz (f)
 ESP ( "cmd=freq&val=" + Trim(Str$(f)))
End Sub
Sub Delay ( ByVal ms)
 t = Timer
 DoEvents
 While Timer < (t + ms / 1000 )
  DoEvents
 Wend
End Sub
```

PWM am Ausgabe Pin funktioniert nur, wenn eine quasi analoge Spannung ausgegeben wird. Mit *Led(1)* führt die ESP-Server-Routine ein *io(pwo,15,512)* aus und damit ist ein Rechtecksignal vorhanden. Dieser Aufruf ist vor der Aufnahme der Kurve erforderlich und ist in der *Sub Resonanz()* entsprechend eingebaut.

```
Sub LED (an)
```

```
  ESP ( “cmd=led” + Trim(Str$(an)))
End Sub
Sub Resonanz ()  'ESP/ADC/PWMFREQ: Pin 15 RGB RED
  LED ( 1 )
  Range( “a:c” ).ClearContents
  Zeile = 2
  For f = 100 To 1400 Step 10
   Frequenz f
    Delay 333
    Cells(Zeile, 1 ) = f
    Cells(Zeile, 2 ) = UIN( 1 )
    Zeile = Zeile + 1
  Next f
  LED ( 0 )
End Sub
```

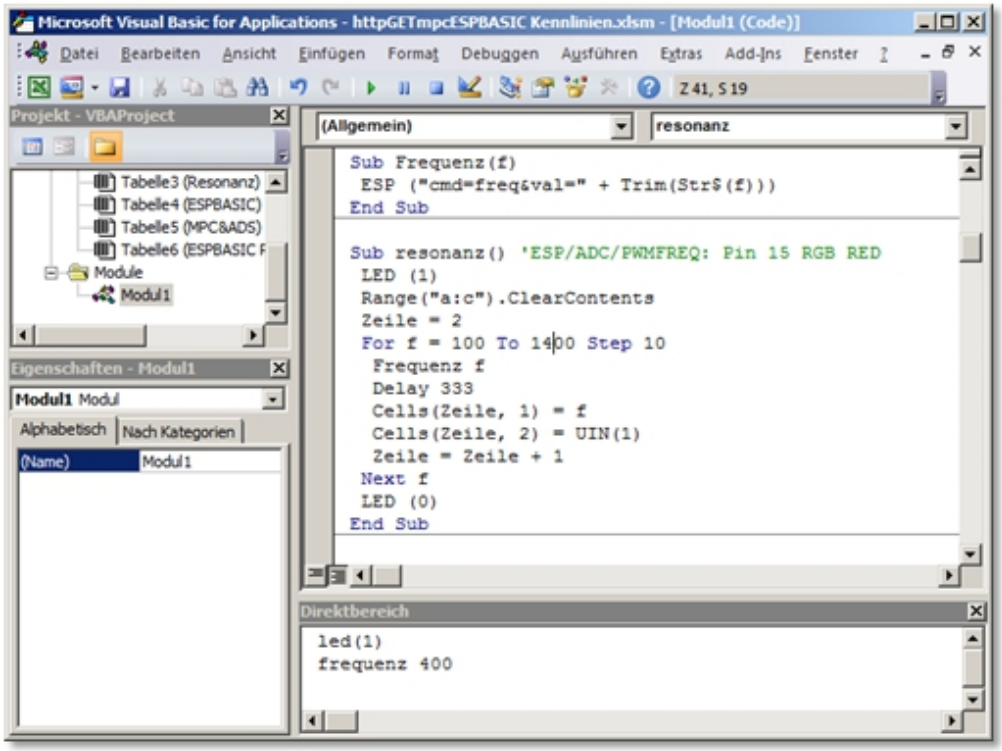


Abbildung 4-37: VBA meets ESPBASIC, ein Blick in den Editor von Excel

Die Frequenz läuft von 100 Hertz bis 1400 Hertz im Intervall 10 Hertz. Das kurze Listing sollte selbsterklärend sein. Die Wartezeit von 333 ms ist rein willkürlich gewählt. Da im *Delay* ein *DoEvents* eingebaut ist, kann der Lauf jederzeit gestoppt werden.

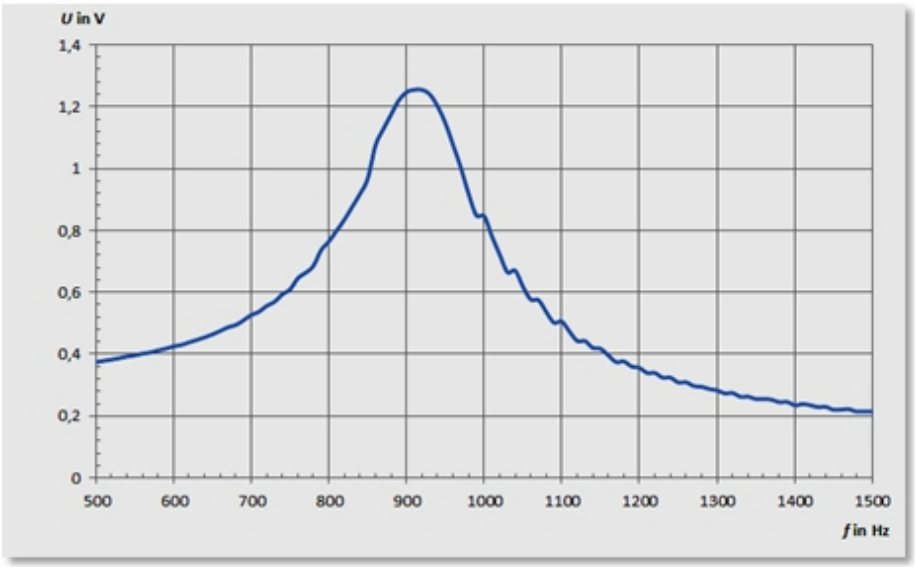


Abbildung 4-38: Resonazkurve mit ESPBASIC

Mit einer Resonanzfrequenz von 910 Hz ergibt sich, nach Anwendung entsprechender Formeln, eine Induktivität von 93 mH. Schaut man sich das Diagramm unter 500 Hz an, so wird deutlich, dass hier ein Rechteck als Speisung dient. Die Stufen im rechten Bereich sind vermutlich darauf zurück zu führen, dass die Ausgabefrequenz bei höheren Werten bei unterschiedlichen Ansteuerungen gleich bleibt. Diesen Effekt kann man mit einem Piezo-Beeper auch akustisch wahrnehmen.

Mit RFO-BASIC können solche Messwerte auch direkt auf dem Taschentelefon gemessen und dargestellt werden. Die Darstellung erfolgt dann ebenfalls in Excel, jedoch in der kostenlosen Version für z. B. Android Smartphones. Abschnitt 4.10 zeigt drei Varianten zum Thema Messen mit dem Android-Smartphone.

4.6.11 Steuerbare Spannungsquelle PWM

Ein PWM-Signal ist ein Rechtecksignal mit veränderbarer Impulsdauer bei fester Periodendauer. Der arithmetische

Mittelwert der Ausgangsspannung entspricht diesem Verhältnis bezogen auf den Maximalwert. Die Anweisung *io(pwo,15,255)* sorgt z. B. dafür, dass an einem ESP8266 an Pin 15 ein Rechtecksignal mit einer Frequenz von 1 kHz kontinuierlich ausgegeben wird, die Periodendauer also 1 ms beträgt. Dabei ist das Signal 255 von 1023, also 25% der Periodendauer an und 75% aus. In Zeiten ausgedrückt ist der Pin 250 µs HIGH und 750 µs LOW. Ein Multimeter in Stellung DC zeigt korrekt ¼ der Spitzenspannung an. Bei 3,3 Volt am ESP8266 entspricht das 0,825 Volt und eine LED würde 1000 mal pro Sekunde kurz an und ausgeschaltet; sie scheint gedimmt zu sein.

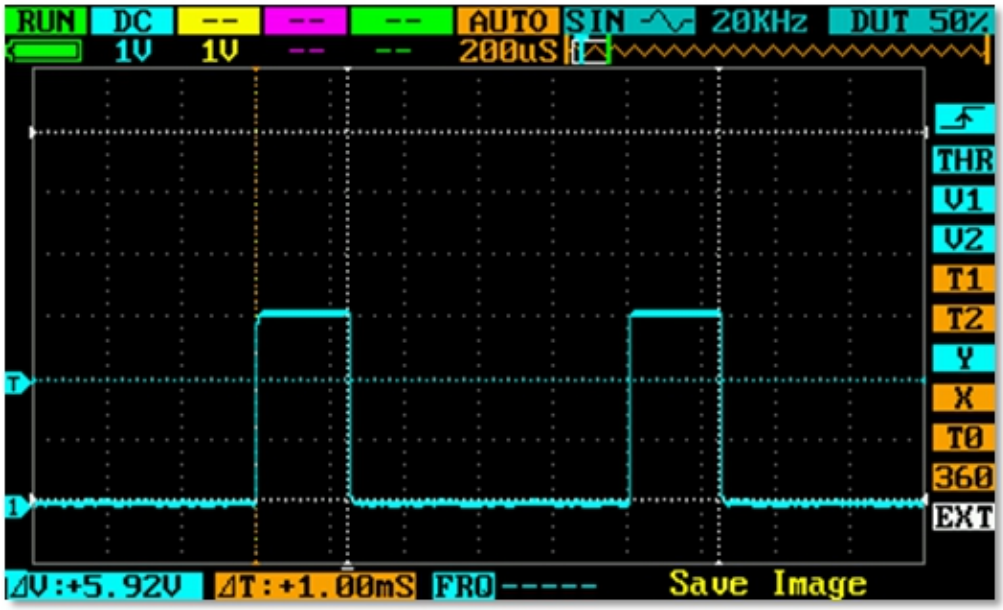


Abbildung 4-39: PWM-Signal auf dem Oszilloskop

Bei der Messung eines solchen Signals mit einem Analogeingang liegt der Messwert, je nach Zeitpunkt der Messung, auf HIGH oder LOW. Wenn die Frequenz des PWM-Signals unabhängig von der Messfrequenz ist, sollte durch Mittelwertbildung dieser Messungen wieder der arithmetische Mittelwert zu erhalten sein. Der folgende Aufbau untersucht diese Annahme mit etwa 30 Messungen, um daraus den Mittelwert zu bilden.

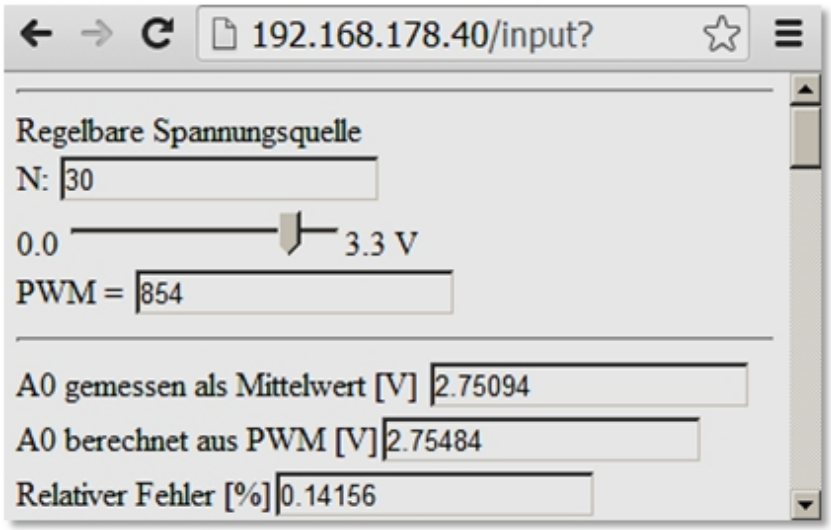


Abbildung 4-40: Gemitteltes Ergebnis von 30 Messungen und Fehler

Mit dem Programm und der dargestellten Oberfläche lassen sich diese Annahmen untersuchen. Die Anzahl *N* lässt sich manuell ändern, der Slider stellt die „Analogspannung“ ein, der PWM-Wert erscheint darunter. Danach folgen Analyse und Fehlerberechnung aus gemessenem und berechnetem Wert.

Das folgende Listing ist ohne *Convert* und *Config* aufgeführt. Im Anhang 5.3.9 befindet sich eine weitere, komplette Variante, die diese Ergebnisse parallel auch über UDP ausgibt. Es ist gleichzeitig ein Beispiel dafür, wie bei Untersuchungen ohne großen Aufwand die Messdaten per WiFi an andere Geräte übertragen werden können.

```
print “Regelbare Spannungsquelle <br>”
memclear
n = 30
html “N: “
textbox n
aout = 511
html “<br>0.0”
slider aout, 0, 1023
vpb = 0.0001875
html |3.3 V<br>PWM = |
textbox aout
html |<hr>|
html “A0 gemessen als Mittelwert [V] “
```

```
textbox u0
html |<br>|
html “A0 berechnet aus PWM [V]”
textbox ux
html |<br>|
html “Relativer Fehler [%]”
textbox frel
html “<br>Fehler [%]<br>”
timer 5000,[messen]
[messen]
s = 0
io(pwo,15,aout)
for i = 1 to n
 MSB = “C1“‘KANAL A0gnd
 gosub [config]
 gosub [convert]
 s = s + u ‘Messwert abrufen
next i
u0 = s / n
ux = 3.3 * aout / 1023
frel = (ux - u0) / ux * 100
html  ”<br>” & frel
wait
…
[convert]
…
[config]
```

4.6.12 Steuerbare Spannungsquelle RC

Mit einer Kombination aus Widerstand und Kondensator kann ein PWM-Signal in Grenzen analoge Werte annehmen. ESPBASIC benutzt eine PWM-Frequenz von 1000 Hertz, die sich aber per Befehl ändern lässt. Damit kann dann per Software das PWM-Signal der RC-Kombination angepasst werden.

Im folgenden Abschnitt soll das Verhalten einer quasi analogen Ausgabe in Hinblick auf deren Brauchbarkeit im

Zusammenhang mit Kennlinienaufnahmen untersucht werden. Die neue Spannungsquelle ist der Ausgang eines Tiefpasses aus *R* und *C* , wobei der Kondensator die Spannung liefert. Gewählt wurde *R* = 1 kΩ und *C* = 220 µF (nach Abbildung 4-28: Aufbau zur RC Auf- und Entladung). Es ist im Prinzip eine Auf- und Entladung mit hoher (PWM-) Frequenz. Als Analog-Digital-Wandler kommt ADS1115 mit einer Mehrkanalmessung zum Einsatz, da mit diesem Aufbau auch Kennlinien aufgenommen werden sollen.

Das Witty Cloud Modul speist den ADS1115 mit 5 Volt V_{DD} und Gnd. Pin GPIO00 (SDA) und GPIO02 (SCL) bedienen den I2C-Bus. Mit dem I2C-Scanner aus Abschnitt 0 erfolgt ein kurzer Check der Adresse: *Finde Adresse 0x48 (72)* . Damit sollte das Listing der Einkanalmessung von weiter oben unverändert funktionieren. Nach Aufbau der RC-Schaltung, die von einem freien Digitalausgang z. B. GPIO 05 mit dem PWM-Signal gespeist wird, erhält das Listing zu Beginn noch einen *Slider* und eine *Textbox* mit der Variablen *aout* als Analog-Ausgang, um die Spannung manuell zu steuern und das Ergebnis zu betrachten.

```
slider aout,0,1023
textbox aout
wprint “<hr>”
```

Ein zweiter Timer dient der Steuerung. *Timercb 200, [cb]* ruft die Zeilen beim Branch *[cb]* im 0,2 Sekunden Intervall auf, während der *Timer* der Messroutine nur noch zweimal pro Sekunde aufruft. Der Teil des Listings, der verändert ist:

```
print “ADS1115 A0/GND PWM RC<br>”
i2c.setup(0,2)
vpb = 0.0001875’ Gain 0.67
slider aout,0,1023
textbox aout
wprint “<hr>”
textbox bits
wprint ” Bits (“&vpb&” Volt/Bit)<br>”
textbox u
wprint ” U/V<br>-5……0……+5<br>”
```

meter u,-5,5

timer 500 ,[messen]

timercb 200,[cb]

wait

[cb]

io(pwo,5,aout)

return

[messen]

…

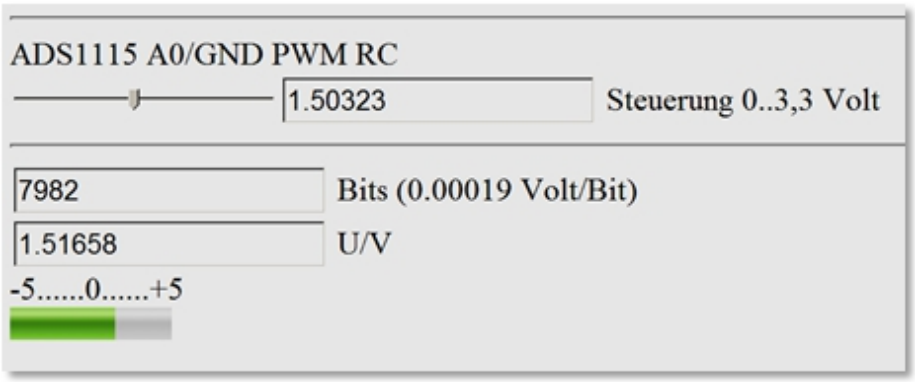


Abbildung 4-41: Manuell steuerbare Gleichspannungsquelle

Das Ergebnis zeigt eine Ausgangsspannung, die so stabil ist, dass erst an der dritten Nachkommastelle Schwankungen zu beobachten sind. Um die Stabilität dieser unbelasteten Spannungsquelle zu verdeutlichen kann mit der kleinen Änderung in der Messroutine, die Nachkommazahl auf zwei Stellen reduziert erscheinen:

…

uu = bits * vpb

u = int(uu*100)/100

…

Weitere Untersuchungen mit manuell veränderter PWM-Frequenz erscheinen daher nicht nötig, obwohl sie sich mit einer weiteren Variablen-Slider-Textbox-Kombination in wenigen Zeilen realisieren lässt.

4.7 12-Bit-Digital-Analog-Wandler - MCP4725

PWM ist schnell und preiswert, jedoch für analoge Messungen eher die zweite oder dritte Wahl. Eine analoge und regelbare Spannungsquelle erhält man mit dem preiswerten Digital-Analog-Wandler MCP4725.

Abbildung 4-42: MCP4725 DAC

Dieser Baustein fügt sich mit vier Verbindungen in einen Messaufbau ein, wobei neben der üblichen Spannungsversorgung die Kommunikation über I2C abläuft. Da ESPBASIC dieses Protokoll eingebaut hat, reichen wenige Zeilen, um eine gesteuerte Spannungsquelle zu erhalten. Der fünfte Anschluss ist der Ausgang, der dann eine zu untersuchende Schaltung oder ein einzelnes Bauelement mit einer Auflösung von 12 Bit versorgen kann. Der Maximalwert der Ausgabe bestimmt Vcc des Wandlers, in diesem Umfeld also 3,3 oder 5 Volt.

Mit einem angegebenen maximalen Ausgangsstrom von etwa 25 mA lassen sich z. B. Kennlinien von passiven Bauteilen aufnehmen. Neben den Klassikern Silizium- und Germaniumdiode, bieten sich aber auch die farbigen Leuchtdioden mit ihren unterschiedlichen Leitungsverhalten an. Ein Hauch von Quantenphysik weht durch den Raum, wenn Schleusenspannung auf Lichtquant trifft und Rückschlüsse aufgrund der verwendeten Atome oder Materialien erfolgen. Falls die Adresse des I2C-Teilnehmers nicht auf der Platine steht, lässt sich diese mit dem I2C-

Scanner aus Kapitel 3 an den Anschlüssen 0 und 2 bestimmen.

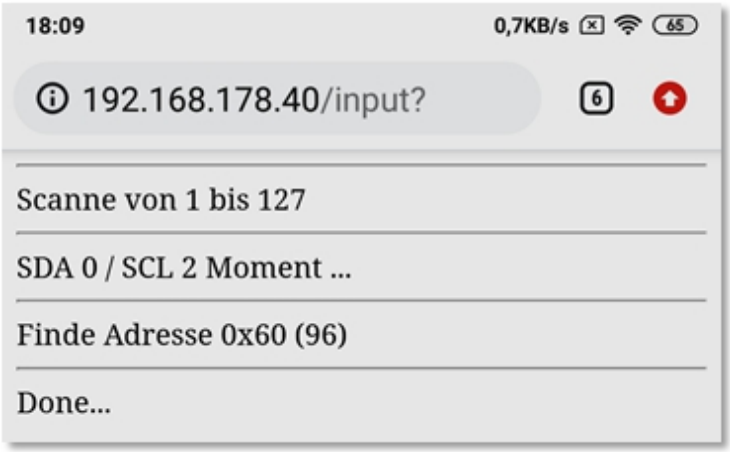


Abbildung 4-43: Diese Platine horcht auf der Adresse 96

Da auch der ADS1115 AD-Wandler an I2C angeschlossen ist, lassen sich, auch ohne den ESP eigenen ADC, Messungen dieser Art durchführen. Da das hier verwendete Modul an dieser Stelle einen lichtempfindlichen Widerstand angeschlossen hat, der sehr gute Dienste leistet, sind alternative und hochauflösende Eingänge willkommen. Eine recht übersichtliche Verschaltung von ESP8266, ADS1115 und MPC4725 könnte also wie folgt aussehen:

ESP	***ADS***	***MCP***
5,0 V	VDD	VCC
GND	GND	GND
GP00	SDA	SDA
GP02	SCL	SCL

Bei einer 5 Volt Versorgung kann der Ausgang laut Hersteller auch die Steuerung der Ausgangsspannung bis zu dieser Höhe in 4096 Stufen (12 Bit) erfolgen. Bei Vollmond lieferte der ESP allerdings nur 4 Volt, so dass beim ersten Einschalten 2 Volt am Ausgang lagen. Es war die im EEPROM gespeicherte Spannung, die auch nach dem Abschalten erhalten bleibt, so

dass eine programmierbare Spannungsquelle zur Verfügung steht. Zu beachten ist der Maximalstrom von 25 mA.

Um zum ersten Mal mit ESPBASIC eine analoge Ausgangsspannung zu erhalten, reicht das folgende kurze Listing. Es ist die Umsetzung der entsprechenden Arduino-Bibliothek in ESPBASIC. Die erste Zeile bestimmt die Anschlusspins, wie schon beim I2C-Scanner. Die Variable *dac* erhält den Maximalwert, um zu testen welcher Maximalspannung dies entspricht. Der Ablauf ist zu vergleichen mit dem ADS1115-Baustein, nur dass hier nichts zurück gelesen werden muss. Das Listing würde, durch Wegfall der Hilfsvariablen *h* und *l* auf ganze 6 Zeilen reduziert, auch funktionieren.

```
i2c.setup(0,2)
dac = 4095  'Max
h = dac / 16
l = dac % 16 << 4
i2c.begin(96)
i2c.write(64) 'DAC 0x40
i2c.write(h)
i2c.write(l)
i2c.end()
End
```

Nach dem Aufruf zeigte ein Multimeter 4,00 Volt. Ein Test mit der 3,3 Volt Spannung lieferte 3,3 Volt am Ausgang. Vermutlich klemmte etwas beim 5 Volt Spannungsregler. Diese Zeilen lassen sich zur Spannungsausgabe in einem Unterprogramm aufrufen mit vorheriger Zuweisung eines entsprechenden Wertes an die Variable *dac* . Bei Ausgabe aller 4096 Stufen in einer *For-Next* -Schleife, wie bei der Arduino-Bibliothek wird deutlich, dass diese Sprache eine Interpreter-Sprache ist. Ein Sinusgenerator in ESPBASIC macht aufgrund der Geschwindigkeit wenig Sinn. Aber die Einfachheit und Kürze der Anwendung neuer I2C-Hardware ist beeindruckend. Das Listing der beschleunigten Rampe bzw. Sägezahns:

```
i2c.setup(0,2)
for dac = 0 to 4095 step 32
```

```
 gosub [aout]
next dac
End
[aout]
i2c.begin(96) ‘ADR 0x60
i2c.write(64) ‘DAC 0x40 EEPROM 0x60
i2c.write(dac / 16)
i2c.write(dac % 16 << 4)
i2c.end()
Return
```

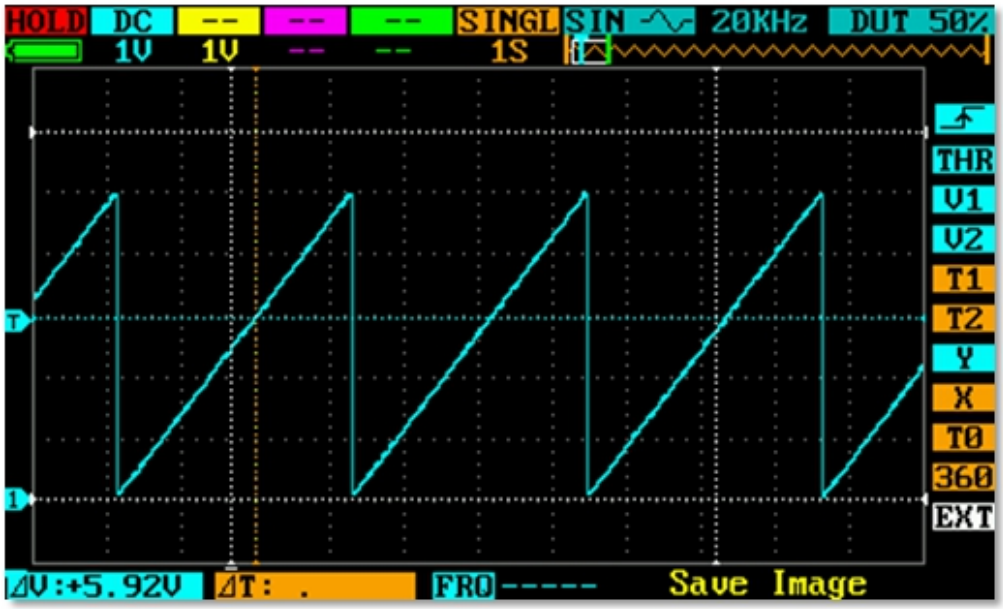


Abbildung 4-44: Sägezahn aus dem MPC4725

Um eine Messung ohne Multimeter zu erhalten, kommt der ADC des ADS1115 mit zwei Eingängen zum Einsatz. Mit der Grundverstärkung 2/3 soll mit Eingang A0 und A1 die Spannung wieder zurück gewandelt werden. An den I2C-Leitungen befinden sich nun zwei Geräte mit zwei Adressen, wie ein weiterer Scann zeigt:

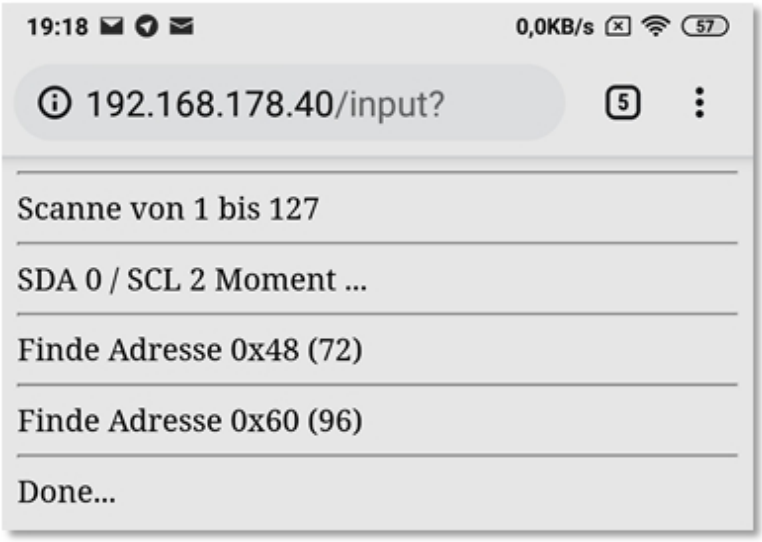


Abbildung 4-45: Zwei Teilnehmer reagieren an den I2C-Leitungen

Einer Anwendung beider Komponenten steht nichts mehr im Weg.

4.7.1 KENNLINIENWERTE IM BROWSER

Als Messschaltung soll eine Reihenschaltung bestehend aus einem Bauelement und einem bekanntem Messwiderstand dienen. Als Bauteile sollen Widerstände und Dioden zum Einsatz kommen, um deren Kennlinien zu bestimmen.

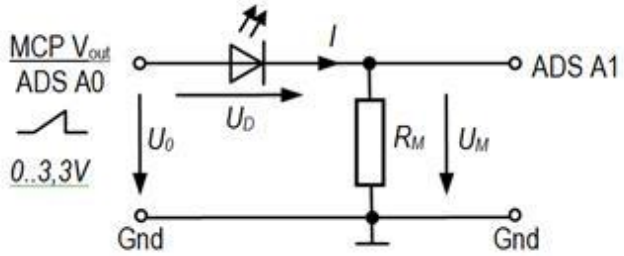


Abbildung 4-46: Messschaltung Kennlinienaufnahme

Der Widerstandswert des R_M lässt sich mit einem Multimeter ausmessen und dann fest einbauen. Die Kennlinie besteht aus einem Strom-Spannungs-Diagramm des Bauteils und soll im Bereich 0 bis 10 mA und 0 bis 3,5 Volt zu erkennen sein. Analogeingang A0 misst die vom DAC erzeugte Analogspannung als U_0 der Gesamtschaltung. Analogeingang A1 des ADS1115 nimmt die Spannung zwischen Bauelement und Messwiderstand als U_1 auf. Die gesuchten Werte für Strom und Spannung berechnen sich nach:

$$I = \frac{U_1}{R_M} \quad , \qquad U = U_0 - U_1$$

Der geringe Strom wird in Milliampere umgerechnet zur Anzeige gebracht. Das folgende Programm gestattet eine manuelle Untersuchung des Schaltungsaufbaus, um mögliche Fehlerquellen zu eliminieren. Die Kennlinienaufnahme soll später automatisch erfolgen.

Ein *Slider* steuert die Spannungsquelle DAC und mehrere *Textboxes* zeigen die mit ihnen verknüpften Variablen. Die unteren beiden Werte sind die gesuchten Größen des Bauelements. Der *Timer* ruft zweimal pro Sekunde die Steuerung und die Messung auf. Unterhalb der Timer-Routine bei *[tm]* stehen die Programmteile zur Kommunikation der erweiterten Hardware am I2C-Bus, die weiter oben in ähnlicher Form bereits benutzt worden sind. Das Gesamtlisting in reinem ESPBASIC ist dadurch etwas länger, sollte aber noch auf ein A4-Blatt passen.

```
i2c.setup(0,2)
rm = 333 ‘Ohm gemessen
Slider dac, 0, 4095
Textbox dac
Textbox u0
Textbox u1
Textbox ud
Textbox id
Timer 500, [tm]
Wait
[tm]
Gosub [aout]
gosub [messen]
wait
[aout]
i2c.begin(96)
i2c.write(64) ‘DAC 0x40
i2c.write(dac / 16)
i2c.write(dac % 16 << 4)
```

```
i2c.end()
Return
[messen]
MSB = “C1” ‘KANAL A0gnd
gosub [config]
gosub [convert]
u0 = u ‘Messwert abrufen
MSB = “D1” ‘KANAL A1gnd
gosub [config]
gosub [convert]
u1 = u
ud = u0 - u1
id = u1 / rm * 1000
return
[config]
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
delay 2
return
[convert]
vpb = 0.0001875
i2c.begin(hextoint(“48”))‘adr
i2c.write(0) ‘CONVERT
i2c.end()
i2c.requestfrom(hextoint(“48”),2)
msb = i2c.read()
lsb = i2c.read()
‘Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits > 32767 then bits = bits - 65536
u = bits * vpb
return
```

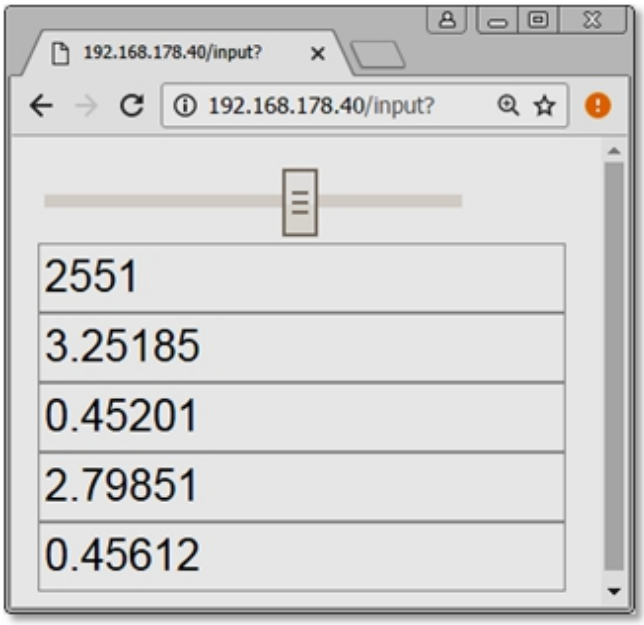


Abbildung 4-47: Manuelle Steuerung mit Slider

Der Screenshot zeigt die Messung an einer grünen LED, die bei 2,8 Volt 0,46 mA fließen lässt. Bezogen auf seine Versorgungsspannung produziert die steuerbare Spannungsquelle mit dem Wert 2551 von 4095 eine Versorgungsspannung für den Messaufbau von 3,25 Volt von denen 0,45 Volt am Messwiderstand abfallen.

4.7.2 KENNLINIEN IN EXCEL ÜBER HTTP

Der ESP ist ein WiFi-Baustein und darum soll die Kennlinienaufnahme drahtlos direkt in Excel über VBA erfolgen, ähnlich wie weiter oben beim RS232-Interface. Hier die Ergebnisse verschiedener „gebrauchter“ LED.

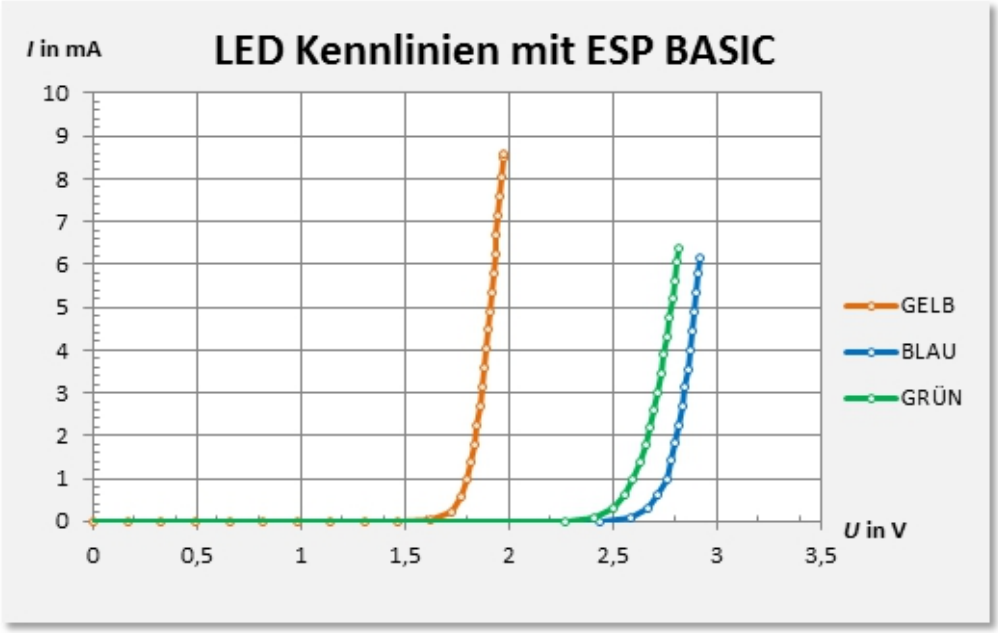


Abbildung 4-48: LED-Kennlinienschreiber direkt in Excel-Diagramm

Dieses Ergebnis erhält man, wenn ESPBASIC als Mess- und Steuerserver fungiert. Es sind nur drei Kommandos notwendig, die der Server behandeln muss:

- *AOUT* – Analogausgabe
- *AIN1* – Analogeingang A0
- *AIN2* – Analogeingang A2

In einem *msg* -Handler in ESPBASIC sieht das z. B. wie folgt aus:

```
memclear
Print “DAC & ADC over WiFi”
i2c.setup(0,2)
u = 1.2345678
msgbranch [br]
wait
[br]
```

```
cm = upper(msgget("cmd"))
va = val(msgget("val"))
MyReturnMsg = "ok"
if cm == "AIN1" then
 Gosub [ain1]
 MyReturnMsg = str(u)
end if
if cm == "AIN2" then
 Gosub [ain2]
 MyReturnMsg = str(u)
end if
if cm == "AOUT1" then
 dac = va
 Gosub [aout]
 MyReturnMsg = str(va)
end if
If cm == "LED1" then Io(po,15,1)
If cm == "LED0" then Io(po,15,0)
if cm == "LDR" then
 MyReturnMsg = str(io(ai))
end if
msgreturn MyReturnMsg
wait
[aout]
i2c.begin(96)
i2c.write(64) 'DAC 0x40
i2c.write(dac / 16)
i2c.write(dac % 16 << 4)
i2c.end()
return
[ain1]
MSB = "C1"
gosub [config]
gosub [convert]
return
[ain2]
```

```
MSB = “D1”
gosub [config]
gosub [convert]
return
[config]
i2c.begin(72)
i2c.write(1)
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
return
[convert]
vpb = 0.0001875
i2c.begin(72)
i2c.write(0)
i2c.end()
i2c.requestfrom(72,2)
msb = i2c.read()
lsb = i2c.read()
bits = msb * 256 + lsb
if bits > 32767 then bits = bits - 65536
u = bits * vpb
return
```

Die LED- und LDR-Abfragen und Steuerungen können auch entfallen, sie dienen nur zur Fehlerfindung, denn ESPBASIC zeigt sich zumindest in der 2MB-Version von Branch 69 etwas störrisch, indem stur behauptet wird, dass kein *Msg* - Handler für *AIN1* existiert. Darum erscheinen die I2C-Routinen teilweise mit direkter Adressangabe um Platz zu sparen. Zumindest funktioniert es hier in dieser Form – übrigens auch mit RFO-BASIC auf dem Android-Smartphone. Wird dieses Programm als *default.bas* gespeichert, so startet es nach einem Crash oder Neustart nach 30 Sekunden automatisch, wenn das in den Settings gewünscht ist.

Im VBA-Modul in Excel gibt es neben der ESP-Routine, wie an anderer Stelle schon benutzt, zwei neue Interface-Routinen für die Kennlinien-Aufnahme „Over The Air“:

```
Function ESP ( ByVal Cmd$)
Set http = CreateObject( "WinHttp.WinHttpRequest.5.1" )
    http.Open "GET" , "http://192.168.178.40/msg?" + Cmd$
    'http.Open "GET", "http://192.168.4.1/msg?" + Cmd$
   http.send
    ESP = Replace(http.ResponseText, "." , "," )
    Set http = Nothing
End Function
'MPC & ADS
Sub AOUT ( ByVal nr%, ByVal wert%) 'MPC 1, 0..4095
 ESP ( "cmd=aout" + Trim(Str$(nr)) + "&val=" + Trim(Str$(wert)))
End Sub
Function UIN ( ByVal nr) As Double ' U in Volt
  UIN = CDbl (ESP( "cmd=ain" + Trim(Str$(nr))))
End Function
```

Mit *aout(1,2000)* im Direktbereich stellt man etwa die halbe Ausgangsspannung am DAC ein und mit *uin(1)* liefert der ESP diesmal die Spannung direkt in Volt von Eingang A0 des ADS1115, weswegen die Routine als Double deklariert ist. Ein *uin(2)* liefert den Wert von A1.

Nun folgt die VBA-Routine, die mit diesen drei Befehlen Kennlinien aufnehmen kann. VBA unterscheidet nicht zwischen Groß- und Kleinschreibung.

```
Sub xyDAC () 'MPC/ESP/ADC ONLY Kennlinie
  Range( "a:c" ).ClearContents
  zeile = 2
  dac = 0 : Rm = 333
  Do
    AOUT 1 , dac
    Cells(zeile, 3 ) = dac
```

```
    Delay 50  'siehe weiter oben
    u1 = UIN( 1 )  ' u gemessen
    u2 = UIN( 2 )  ' uR gemessen an Rm
    Ud = u1 - u2  ' Diodenspammung
    ID = (u2 / Rm) * 1000  'Diodenstrom/mA
    Cells(zeile, 1 ) = Ud
    Cells(zeile, 2 ) = ID
  DoEvents
  calculate
    zeile = zeile + 1
    dac = dac + 128
 Loop Until (dac > 4000 Or ID > 10 Or Ud > 4 )
 AOUT 1 , 2000
End Sub
```

Die Messung erfolgt bis die *dac* -Variable den Wert 4000 erreicht hat, oder der Strom durch das Bauteil größer als 10 mA ist, oder die Spannung am Bauelement mehr als 4 Volt beträgt. Die Messung beginnt in Zeile 2, damit in der ersten Zeile Beschriftungen möglich sind. Eine Erhöhung des *dac* -Wertes um 128 hat sich als praktikabel erwiesen, wodurch etwa 33 Messpunkte aufgenommen werden.

4.8 SINUSGENERATOR AD9850 - 30 MHZ

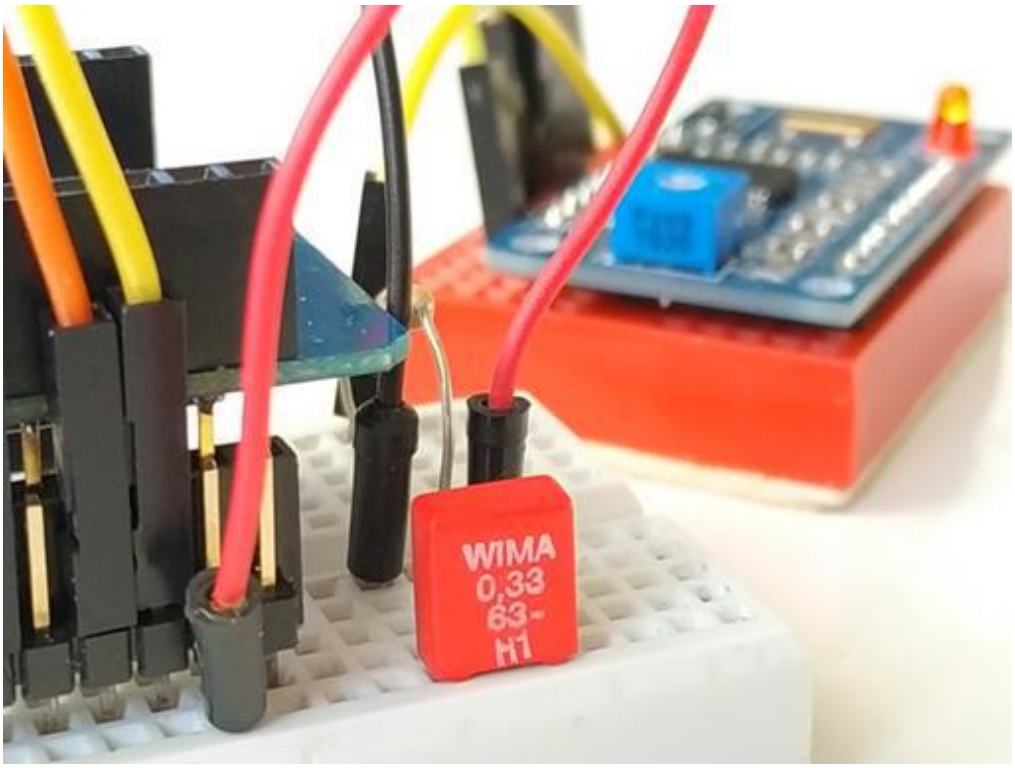


Abbildung 4-49: Zusammenspiel ESP8266 D1 mini und AD9850

4.8.1 MESSUNGEN MIT SINUSSPANNUNGEN

Mit Hilfe des Sinusgenerators AD9850 und der schnellen Kommunikation über SPI, können von ESPBASIC aus Untersuchungen erfolgen, die in einer Compiler-Programmiersprache sich sehr umständlich und zeitintensiv gestalten würden. Ein AD9850 deckt den gesamten Frequenzbereich von 1 bis 30 MHz ab und ist damit interessant für Untersuchungen mit Sinusgrößen. Gegenstand einer Untersuchung soll eine RC-Passschaltung sein. Ziel ist die automatische Bestimmung der Kapazität, wenn der Widerstandswert vorgegeben ist.

4.8.2 FREQUENZGANG EINER PASSSCHALTUNG

Eine Reihenschaltung von Widerstand und Kondensator nach folgender Abbildung verhält sich im Wechselstromkreis als Tiefpass.

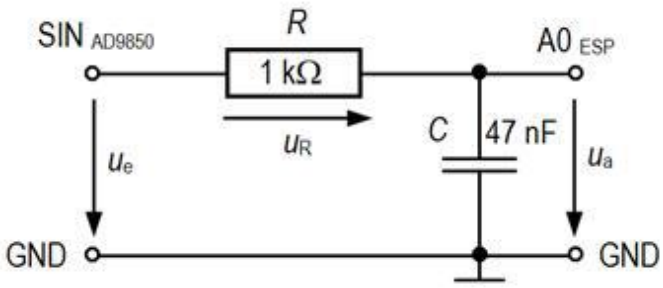


Abbildung 4-50 Kondensator an Sinusspannung - Tiefpass

Das Messsignal am Ausgang ist gegenüber dem Eingang bei hohen Frequenzen geschwächt, niedrige Frequenzen erscheinen ungeschwächt. Der Zusammenhang zwischen Ausgangsspannung und Frequenz stellt der sogenannte Frequenzgang dar. Für konkrete Werte von R = 1 kΩ und C = 0,047 µF ergibt sich der dargestellte theoretische Verlauf.

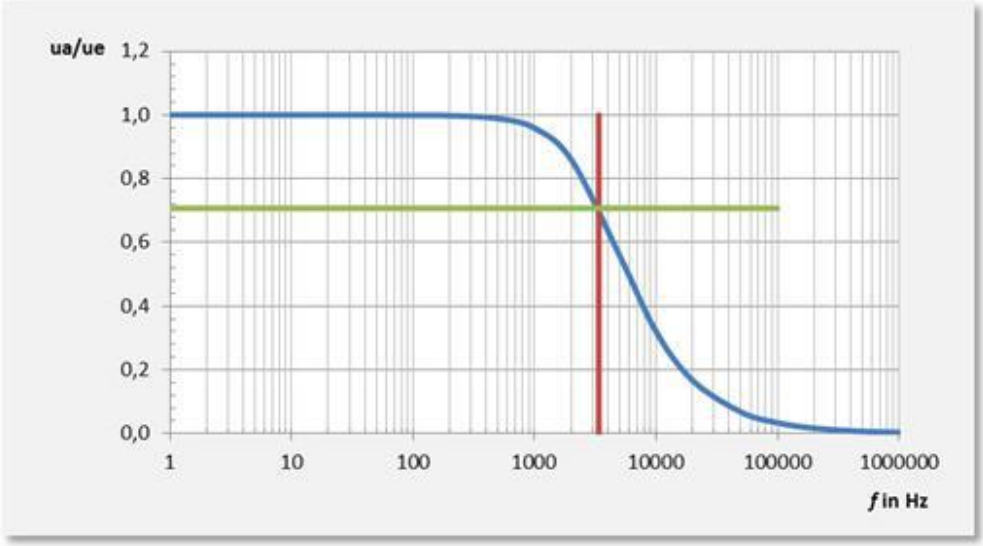


Abbildung 4-51: Frequenzverhalten eines Tiefpasses

Das Verhältnis von Ausgangsspannung zu Eingangsspannung berechnet sich laut Theorie mit Hilfe der folgenden drei Gleichungen:

$$\frac{u_a}{u_e} = cos(\varphi)$$

$$\varphi = arctan\left(\frac{R}{X_C}\right)$$

$$X_C = \frac{1}{2\pi f C}$$

Der Blindwiderstand des Kondensators X_C ist frequenzabhängig und somit auch die Amplitude der Ausgangsspannung am Kondensator. Kennzeichnend für eine solche frequenzabhängige Schaltung ist die Grenzfrequenz f_g, bei der die Werte R und X_c gleich groß sind. Im Zeigerdiagramm entspricht das einem Winkel von 45° zwischen Ausgangs- und Eingangssignal und dem Spannungsverhältnis 0,707. Das Diagramm zeigt Grenzfrequenz und das Verhältnis u_a/u_e bei diesen Bedingungen. Setzt man die beiden Widerstände gleich, so erhält man für diesen Fall die Grenzfrequenz:

$$f_g = \frac{1}{2\pi RC}$$

Abbildung 4-52: Ergebnis nach 34 Messungen

Mit Hilfe des Frequenzgenerators und des Analogeingangs des ESP8266 (ohne LDR) könnte nun dieser theoretische Verlauf überprüft werden. Hier soll jedoch durch messtechnische Annäherung, die Grenzfrequenz bestimmt -, und daraus dann die Kapazität berechnet werden. Die sogenannte schrittweise Annäherung oder sukzessive Approximation liefert im Browser nach 34 Messungen das Ergebnis C = 0,33 µF, wie der Screenshot vom Smartphone zeigt.

4.8.3 Aufbau und Messung

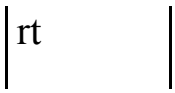
Der Messablauf soll im Browser verfolgbar sein, die Erstellung mit ESPBASIC erfolgen. Die Steuerung des AD9850 und der Ausgabefrequenz erledigen SPI-Aufrufe, die Messung der Ausgangsspannung übernimmt der 10-Bit-Analogeingang eines ESP8266F Moduls.

Der Sinusgenerator liefert eine Gleichspannung in Sinusform. Es gibt also keine Wechselanteile, was die Untersuchung mit Ansätzen aus der Wechselstromtechnik möglicherweise fragwürdig erscheinen lässt. Das Ergebnis der Methode liegt aber trotzdem erstaunlich nah am Aufgedruckten Wert. Der Hardwareaufbau bleibt übersichtlich. Ein Wemos-ESP8266 D1 mini ist mit dem AD9850 wie folgt verbunden:

ESP	***Farbe***	***AD9850***
Vcc (5V)	rt	Vcc
14 (D5)	or	W_CLK
16 (D0)	ge	FO_UF
13 (D7)	gn	DATA
15 (D8)	bl	Reset
GND	br	GND

Die grau unterlegten Verbindungen sind die SPI-Hardwareleitungen des ESP8266 und müssen beibehalten bleiben. Die Messschaltung erhält die sinusförmige Eingangsspannung von Anschluss ZOUT1 des AD9850, der Analogeingang A0 des ESP8266 ist zwischen den beiden Bauteilen angeschlossen und misst die Ausgangsspannung.

Eingang	***Farbe***	***Ausgang***
$ZOUT1_{AD9850}$	ge	
GND	sw	GND



Der Aufbau kommt ohne Hilfsdiode zwecks Gleichrichtung aus, da es sich hier bereits um eine Gleichspannung handelt und der Spannungsbereich ohne Zusatzverstärkung bereits sehr niedrig ist und auch auf eine Wechselspannungsankopplung verzichtet wird. Das hat zu Folge, dass die Bestimmung der Ausgangsspannung nicht als einfache Gleichspannungsmessung erfolgen kann. Die Messroutine in ESPBASIC nimmt *n* Messungen vor und versucht daraus den maximalen und den minimalen Spannungswert der sinusförmigen Gleichspannung zu erhalten. Die Differenz entspricht dann etwa einer Spitzen-Spitzen-Spannung U_{ss} einer Wechselspannung. Die Anzahl *n* ergibt sich experimentell – in ESPBASIC kein Problem - und beträgt hier 50. Ein Messpunkt besteht also aus 50 einzelnen Analogmessungen an A0. In ESPBASIC sieht das so aus:

```
[messen]
max = 0
min = 5
for j = 1 to 50
 aa = io(ai) / 310
 if aa > max then max = aa
 If aa < min then min = aa
Next j
a = (max - min)
…
```

Die Variable *aa* enthält den 10-Bit-Analogwert des Eingangs A0 (*ai*) umgerechnet auf den Bereich 3,3 Volt. Die Variable *a* enthält nach 50 Messungen quasi die Spannung U_{ss} . Da nur ein Analogeingang zur Verfügung steht und ein ADS1115 mit I2C-Ansteuerung viele Zeilen und Zeit bräuchte, erfolgt die Bestimmung der Eingangsspannung *a0* in dieser Anwendung durch einmalige Messung der Ausgangsspannung bei der niedrigsten Frequenz zu Beginn der Messreihe. Als Ergebnis liefert die Messroutine das Verhältnis u_a / u_e bzw. *a /a0* oder *u/u0* .

```
…
if a0 < 0 then a0 = a
c = (a / a0)
u = c
cnt = cnt + 1
return
```

4.8.4 Sukzessive Approximation

Mit der schrittweisen Annäherung soll Messung und Auswertung einer Untersuchung quasi parallel ablaufen. Die Aufnahme einer Messreihe, anschließende Darstellung und Auswertung soll einer automatisierten Messwertbestimmung weichen, ähnlich der Komponententester oder Multimeter in Stellung Automatik. Die Messung soll zwischen 100 Hz und 10 MHz den Wert des Spannungsverhältnisses bei der Grenzfrequenz ermitteln. Ausgehend von der Anfangsfrequenz liegt dieser Punkt weiter rechts auf der Frequenzachse. Das Programm teilt den Frequenzbereich linear in *N* Abschnitte und sucht in dem jeweiligen Bereich die erste Unterschreitung des Verhältnisses von u_a / u_e = 0,707 -, also die Überschreitung der Grenzfrequenz. Ein Wert *N* = 20 liefert in diesem Umfeld praktikable Werte. In ESPBASIC erfolgt der Aufruf *[apx]:*

```
[apx]
C = “mal sehen…”
stat = “suche Grenzfrequenz…”
st = (mx - mn) / N
ff = mn
f = mn
```

…

In *mn* und *mx* sind die jeweiligen Grenzen der Frequenzuntersuchung gespeichert und betragen zu Beginn *mn* = 100 und *mx* = 10000000. Die schrittweise Erhöhung erfolgt um *st.* Die anfänglich benutzte *For* /*Next* -Schleife ist nun eine *Goto* -Konstruktion, die in anderen Programmiersprachen der höheren Art ungern gesehen wird, allerdings nichts anderes darstellt als eine *Do* /*Loop* -Struktur mit einem *ExitIf* . Die

Variable *f* enthält die aktuelle-, die Variable *ff* die vorige Frequenz (eine Art Schleppzeiger). Die Routine *[apx]* setzt sich fort mit:

```
[wdh]
 gosub [set]
 gosub [messen]
 if u <= 0.707 then
  mx= f 'neue Grenzen
   mn = ff
  goto [exit]
 endif
 ff = f
 f = f + st
goto [wdh]
[exit]
…
```

Das Unterprogramm *[set]* stellt die Frequenz *f* am AD9850 ein, das Unterprogramm *[messen]* liefert das Spannungsverhältnis. Die Frequenz wird schrittweise erhöht bis bei Überschreitung der Grenzfrequenz, also bei einem Verhältnis unter 70,7 % die Variablen, für Anfang *mn* und Ende *mx* des zu untersuchenden Frequenzbereiches entsprechend neue Werte zugewiesen bekommen, um eventuell einen neuen Durchgang zu starten, was die Zeilen an der Stelle *[exit]* entscheiden.

Am Ausgang *[exit]* zeigt sich, ob die nächste Annäherung erforderlich ist und *[apx]* erneut durchlaufen werden muss, oder ob der Frequenzbereich soweit eingeengt ist, dass untere und obere Frequenz im ganzzahligen Teil übereinstimmen und somit die Ermittlung der Grenzfrequenz erfolgreich beendet werden konnte, oder die Anzahl der Messpunkte den Wert 300 überschreitet, was auf ein Versagen der Routinen hinweist. In ESPBASIC lassen sich die Dinge schnell ändern und anpassen. Die Zeilen bei *[exit]* lauten:

```
[exit]
if int(mn) == int(mx) then
 stat = int(mx)&" Hz gefunden"
```

```
 C = 1000000 / (2 * pi * mx * R)
 wait
endif
if cnt < 300 then goto [apx]
stat = "Kein Ergebnis :-("
wait
```

Die Kapazität errechnet sich aus der Grenzfrequenz und kommt im Textfeld *C* zur Anzeige.

4.8.5 Ansteuerung des AD9850

Die Ansteuerung des Frequenzgenerators in ESPBASIC ist, dank der SPI-Unterstützung, kurz. Mit *spi.setup(1000000,0,0)* erfolgt die Initialisierung für diesen Baustein (vgl. Abschnitt 3: SPI). Die eigentliche Frequenzsteuerung erfolgt im Unterprogramm *[set]* , welches die Variable *f* und *p* auswertet und daraus die Werte für die SPI-Übertragung errechnet. Der Anschluss *FO* entspricht dem FO_UF am AD9850 und dem GPIO 16 (D0) am ESP. Die Ansteuerung erfolgt nach Angaben des Herstellers Analog Devices in der Application Note AN-1070.

```
[set]
dp = f * 4294967296 / 125000000
ph = p << 3
for byte = 1 to 4
 b = dp and 255
 spi.byte(b)
 dp = dp >> 8
next byte
spi.byte(ph)
io(po,FO,1)
io(po,FO,0)
return
```

Das Gesamtlisting unter 5.3.11 enthält noch viele GUI-Elemente für die Browser-Darstellung und lässt das Programm etwas unübersichtlich erscheinen. Auch ist dort noch die alte, theoretische Messroutine *[messenalt]* enthalten, die es erlaubt,

auch ohne AD9850-Hardware in ESPBASIC eine Annährung zu versuchen.

4.8.6 REGELUNGSTECHNIK

Soll dieses Messverfahren einem regeltechnischen Modell zugeordnet werden, so ergäbe sich der folgende Zusammenhang.

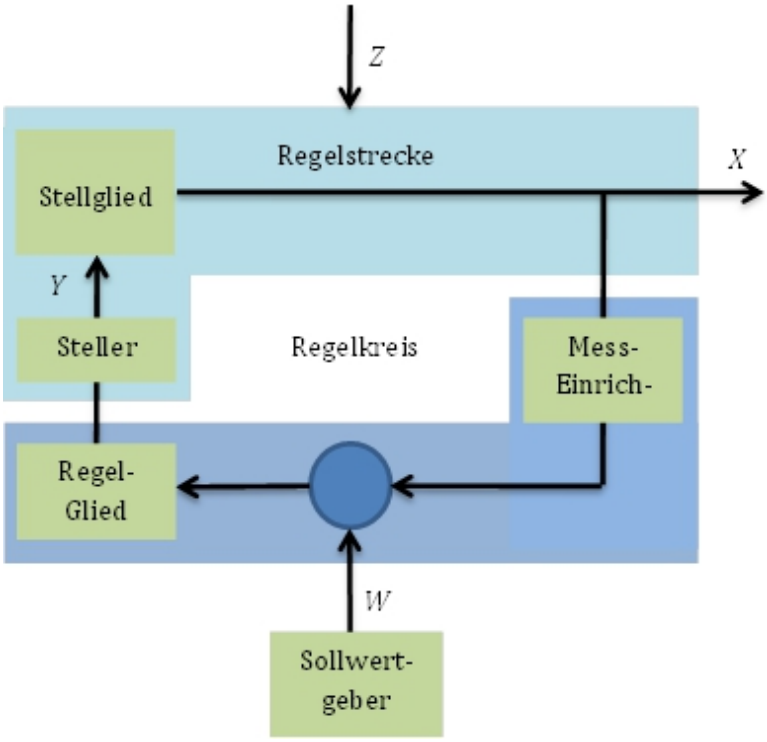


Abbildung 4-53: Regelungstechnisches Äquivalent der Annährung

Die Regelstrecke besteht aus der zu untersuchenden Schaltung aus Widerstand und Kondensator, welche ein bestimmtes Frequenzverhalten aufweist und durch den Labor-Aufbau einigen von außen wirkenden Störgrößen *Z* ausgesetzt ist. Sollwert *W* entspricht dem Spannungsverhältnis von 70% und das Stellglied ist der AD9850-Baustein, der die Stellgröße *Y* verändert. Die SPI-Ansteuerung dient als Steller und der Analogeingang A0 des ESP als Messeinrichtung, die die Regelgröße *X* erfasst. Das Programm stellt dann das Regelglied dar. Ob dies als ein stetiger Regler oder als ein 20-Punkt-Regler anzusehen ist, kann man diskutieren -, muss man aber nicht.

4.8.7 FAZIT

Das fertige Programm erlaubt die Bestimmung von Kapazitäten, wenn die Grenzfrequenz im vorgegebenen Rahmen bleibt. Die Beobachtung im Browser und am Oszilloskop gestaltet sich für Interessierte unterhaltsam. Die Möglichkeit Änderungen am Programm vorzunehmen, die sofort umsetzbar sind, laden zum Experimentieren ein und lassen Untersuchungen zu, die sonst an einer zu hohe Schwelle bezüglich Entwicklung und Aufwand scheitern würden. Der Versuchsaufbau besteht aus einer Powerbank, dem ESP8266 und einem AD9850 und passt fast in die Hosentasche, wenn da nicht schon das Smartphone wäre mit dem die ganze Programmierung und Durchführung erfolgt.

Abbildung 4-54: Grenzfrequenz Annährung im Browser

Das etwas umfangreichere Listing zur Erzeugung der obigen Darstellung und der eigentlichen Messung ist im Anhang 5.3.11 aufgeführt.

4.9 ARDUINO ALS I2C-SLAVE

Nicht jede Hardware-Erweiterung lässt sich mit ESPBASIC ansprechen. Es kann auch sein, dass die Umsetzung in ESPBASIC aus einer Vorlage oder aus dem Datenblatt zu umständlich, zu groß, oder zu zeitaufwändig ist. Falls das Problem bereits mit einem Arduino gelöst wurde, könnte man den Arduino als Knecht bzw. Sklaven – Slave – einsetzen und die Hardware indirekt ansprechen. In der Arduino-IDE findet man fertige kleine Beispiele unter *Datei/Beispiele/Wire/slave_receiver* und *slave_sender* die einen Arduino als I2C-Slave mit der Adresse 08 konfigurieren. Die Kommunikation zwischen ESPBASIC und dem Arduino erfolgt dann über zwei die I2C-Leitungen, die möglicherweise auch schon von anderen Teilnehmern benutzt werden. Der Arduino ist dann der Lieferant oder das Stellglied für andere Hardware. Dies können Radarsensoren oder Ultraschallsensoren sein, die im µs-Bereich abfragen. Aber auch für Frequenzgeneratoren existieren fertige Sketche für den Arduino mit SPI-Ansteuerung, die sogar seriell über RX/TX gesteuert werden könnten.

Um z. B. einen Ultraschallsensor PING aus dem Artikel „Arduino als Zollstock“ (Google) an ESPBASIC anzubinden, bietet sich ein Arduino als I2C-Slave an. Fügt man die Beispiele aus der IDE für den Arduino als I2C-Sklaven zusammen, so dass im selben Sketch gesendet und empfangen werden kann, steht einem Test nichts mehr im Weg. Für BASIC-Puristen könnten fertige HEX-Dateien dieses Sketches zur Verfügung gestellt werden. Der Ultraschallsensor soll dabei stellvertretend für andere Komponenten stehen.

In einem Szenario mit ESP D1 Mini und Android-Smartphone wäre folgende Vorgehensweise denkbar:

- I2C-Slave-Sketch anpassen mit vorhandenen Zeilen der speziellen Hardware
- Sketch als Binärdatei speichern und auf ein Smartphone übertagen, um ihn bei Bedarf hochzuladen.

(Sammlung mehrerer Hardwaresketche möglich)

- D1 Mini Erweiterungsboard mit Arduino Mini Pro oder Nano bestücken (stapelbarer Aufbau möglich)
- *ArduinoDroid* starten und gewünschte Binärdatei (HEX) übertragen
- ESPBASIC über I2C-Adresse 8 mit dem Arduino kommunizieren, der Ein-und Ausgaben weiterleitet.

Es entsteht so ein modulares Hard- und Softwaresystem für verschiedene Anwendungsfälle.

4.9.1 Ultraschall über I2C

Eines der ersten Arduino-Projekte war zu seiner Zeit „Der Arduino als Zollstock“ ***http://www.hjberndt.de/soft/ardping.html*** . Zum ersten Mal wurde damals klar, wie einfach es geworden war, auch komplexe Hardware mit drei Klicks zusammen zu bauen und zum Funktionieren zu bringen. Die damaligen Ausführungen zum PING-Sensor sind unter der Adresse

https://www.arduino.cc/en/Tutorial/Ping

noch abrufbar.

Abbildung 4-55: PING von www.parallax.com

Das Prinzip besteht aus einer Ein-Draht-Kommunikation über einen Pin (7). Zwei Mikrosekunden Low und fünf Mikrosekunden High und anschließendem Low lösen die Messung im Chip des Sensors aus. Der Pin wird dann als Eingang umgeschaltet und mit *pulseIn* wird gewartet, bis der Sensor die Leitung auf Low legt. *PulseIn* liefert die Zeit in Mikrosekunden, die der Schall gebraucht hat. Damit lässt sich dann über die Schallgeschwindigkeit der Abstand berechnen.

Im Sketch ist das die Funktion *messen(),* die den Abstand in Millimetern liefert.

Der *Sketch in C* ist sehr kurz, da er nur als Schaltstelle und Messknecht dient und soll hier trotz des Hauptthemas kurz gelistet sein:

```
#include <Wire.h>

const int pingPin = 7 ;

int mm,ms =500 ;

void setup ()

{Serial.begin( 115200 );

 Wire.begin( 8 ); // join i2c bus with address #0x08

 Wire.onRequest(requestEvent); // Senden-Routine

 Wire.onReceive(receiveEvent); // Empfangen-Routine

 mm = messen();

 Serial.println(mm); //Test

}

void requestEvent ()

{Wire.write(byte(mm /256 ));

 Wire.write(byte(mm &255 ));

}

void receiveEvent ( int howMany)

{ if (Wire.available())

  {ms = Wire.read();     // receive byte as an integer

   Serial.println(ms);  // print the integer

 }

}

void loop ()

{mm = messen();

 delay(ms *10 );
```

```
}

int messen ()

{ long mm;

 pinMode(pingPin, OUTPUT);

 digitalWrite(pingPin, LOW);  delayMicroseconds( 2 );

 digitalWrite(pingPin, HIGH); delayMicroseconds( 5 );

 digitalWrite(pingPin, LOW);

 pinMode(pingPin, INPUT);

 mm = pulseIn(pingPin, HIGH) * 340 / 2000 ;

 return ( int )min(mm, 1000 );

}
```

Das I2C-Verfahren funktioniert synchron, also zeitgleich. Das bedeutet, dass im Gegensatz zur seriellen Kommunikation über RX/TX, bei Anforderung sofort eine Antwort kommen muss. Bei dem hier zur Anwendung kommendem Messprinzip des PING-Sensors vergeht nach einer Messdatenanforderung jedoch mindestens die Zeit, die der Schall für den Weg benötigt. Dass ist, je nach Abstand unterschiedlich und für I2C zu lang. Auch das sogenannte „Clock-Stretching“ führte erwartungsgemäß nicht zum Erfolg. Aus diesem Grund misst der Sensor in der Hauptschleife *loop()* kontinuierlich. Der Arduino liefert bei Anfrage lediglich den letzten Messwert in mm sofort zurück. Die Messfolge kann vom Master (BASIC) eingestellt werden, wobei z. B. eine 100 einem Messintervall von 1000 ms entspricht. Die Voreistellung steht auf 5 Sekunden, so dass bei Bereitschaft die blaue LED am Sensor ab und zu blinkt.

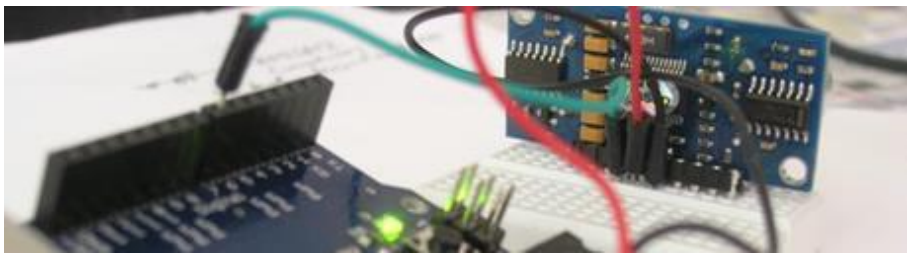


Abbildung 4-56: Arduino als I2C-Sklave

```
Memclear
Print “PING as I2C via Arduino as Slave at 0x08”
ms = 2
i2c.setup(0,2) ‘arduino an 0/2
adr = 8
Textbox mm
print “mm “
Meter mm,0,500
Print “0..250 “
Textbox ms
Slider ms,1,250
Print “(200: delay 2000)”
Timer ms * 10,[rx]
Timercb 1000,[tx]
[rx]
i2c.begin(adr)
i2c.requestfrom(adr,2)
h = i2c.read()
l = i2c.read()
mm = (h * 256) + l
i2c.end()
Wait
[tx]
i2c.begin(adr)
i2c.write(int(ms))
i2c.end()
Return
```

Die Verschaltung erfolgt so, dass entweder der Arduino oder der ESP als Spannungsversorgung (5 Volt) für alle dient. Arduino Pin 7 ist der Daten-Pin zum PING, die Leitungen des I2C-Busses sind SDA (Arduino A4, ESP GPIO 00) und SCL (Arduino A5, ESP GPIO 02). Das ESPBASIC-Programm benutzt zwei *Timer* , jeweils zum Senden und zum Empfangen getrennt. Die Empfangs Routine *[rx]* fordert einen Messwert an, der aus zwei Bytes besteht. Auf Seiten des Arduino wird der Abstand in Millimetern mit zwei Bytes in der Routine

requestEvent() übertragen. Das Messintervall ist auf nur ein Byte begrenzt, wodurch sich etwas Denkarbeit ergibt.

4.9.2 FREQUENZGENERATOR AD9850 ÜBER I2C

Zwar nicht nötig, aber möglich: I2C Kommunikation via Arduino zum Sinusgenerator, als weiteres Beispiel einer Arduino-Slave Anwendung.

```
Memclear
Print “AD9950 to I2C via Arduino as Slave at 0x08”
i2c.setup(0,2)
f = 1000000
Textbox f
Button “Set”,[tx]
adr = 8   ‘arduino an 0/2
Wait
[rx]
i2c.begin(adr)
i2c.requestfrom(adr,2)
h = i2c.read()
l = i2c.read()
mm = (h * 256) + l
i2c.end()
Wait
[tx]
t3 = (f >> 24) and 255
t2 = (f >> 16) and 255
t1 = (f >> 8) and 255
t0 = f and 255
i2c.begin(adr)
i2c.write(int(t3))
i2c.write(int(t2))
i2c.write(int(t1))
i2c.write(int(t0))
i2c.end()
wait
```

Der entsprechende *Sketch in C* befindet sich im Anhang 5.3.12.

4.10 Messen mit dem Smartphone

Mit diesem Titel erschien im Jahr 2013 ein kleines Buch, welches mit dem hardwarenahen RFO-BASIC verschiedene Messungen gestattet. Dieses BASIC ist weiterhin im Playstore kostenlos erhältlich, wurde aber aus teilweise technischen Gründen von dem kompatiblen OLI-BASIC abgelöst, fortgesetzt oder ergänzt. Mit Hilfe dieser Sprache lassen sich anwendungsbezogene und drahtlose Messungen durchführen, die auf dem http-Protokoll beruhen. Damit können vom Smartphone aus kabellose Messungen mit alten seriellen Geräten erfolgen, aber auch mit neuer Hardware, wie dem ESP8266 selbst, oder mit der an ihm über I2C oder SPI angeschlossene erweiterte Hardware. In diesem Zusammenhang werden hier folgend drei Varianten einer Kennlinienaufnahme vorgestellt werden. Die aufwendigste Variante macht den Anfang und die Einfachste kommt zum Schluss. In den drei Anwendungen misst das Smartphone über WiFi mit einem

- alten RS232-Interface mit analogen Aus- und Eingängen
- I2C 16-Bit AD-Wandler und I2C 12-Bit DA-Wandler
- 10-Bit Analogeingang des ESP8266 und I2C 12-Bit DA-Wandler

4.10.1 RFO-BASIC mit RS232-Analoginterface

Zunächst soll ein älteres Gerät aus der Modulbus-Serie herangezogen werden. Ziel ist es mit dem Sios-Interface der ersten Generation (RS232) und dem RS232 – WiFi-Konverter aus Kapitel 4.4 eine Dioden Kennlinie aufzunehmen. Dabei kommt der damals noch verbaute echte Analogausgang zum Einsatz. Das Zusammenspiel der Komponenten erfolgt über mehrere Schichten, die im folgenden Bild verdeutlicht werden sollen.

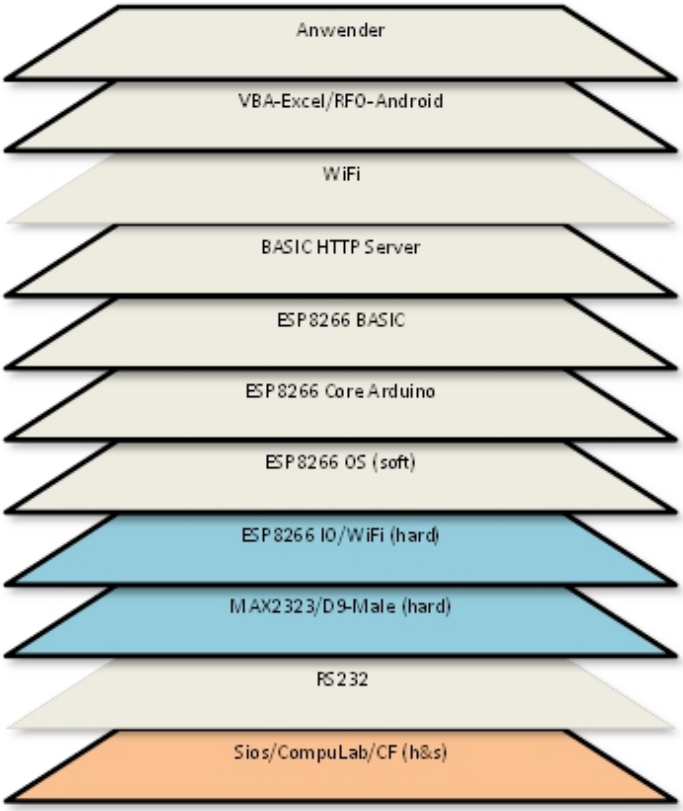


Abbildung 4-57: Schichtenmodell der WiFi-RS232 Strecke

Der Anwender programmiert und misst in RFO-BASIC (Android) oder direkt in VBA/Excel (Windows). Smartphone oder PC müssen WiFi an Bord haben, um sich mit dem ESP8266 verbinden zu können. Im ESP läuft ein in ESPBASIC verfasster sehr kurzer http-Server, der auf das ESP8266-Modul im ESP aufsetzt. Dieser Interpreter ist in C geschrieben und mit der Arduino-IDE und dem ESP-Arduino-Core kompiliert. Der Arduino-Core setzt auf dem Betriebssystem des ESP8266 von AI auf, welches die Verbindung zu WiFi und der Peripherie herstellt. An dieser Peripherie ist der MAX2323 angeschlossen, der dann über den Sub-D Stecker mit dem RS232-Gerät verbunden ist. Falls es zu Störungen im Ablauf kommt, kann man sich nun ein Bild davon machen, wo der Fehler Auftritt und entsprechende Strategien entwickeln.

Um eine Kennlinie mit dem Smartphone messtechnisch zu erfassen und darzustellen sind die folgenden Schritte erforderlich:

- ESP-http-Server als *default.bas* speichern (Anhang 5.3.4: http-Server für das Sios-Interface)
- ESP8266 mit 5 Volt verbinden (Anschlussklemme am Interface) und MAX2323-Konverter mit 3,3 V am ESP8266
- Sub-D Stecker mit Sios verbinden und zum Schluss das Sios mit 12 Volt speisen

Jetzt fährt der ESP8266 hoch und versucht sich mit dem eingestellten Router zu verbinden. Ist kein Router erreichbar, wird ein eigener AP aufgespannt. Nach weiteren 30 Sekunden startet das Programm *default.bas* und zur Kontrolle leuchten die vier rechten LED der Digitalausgänge am Interface. Damit arbeiten die Schichten bis zur oberen WiFi-Schicht.

Abbildung 4-58: RS232-Sios-Interface mit echtem Analogausgang und ESP-Server-Entwurf

Nun folgen die Vorbereitungen auf Seiten der Anwenderhardware. Für das Smartphone gelten folgende Schritte:

- Mit dem Netzwerk des ESP8266 verbinden über Router (192.168.xxx.xxx) oder direkt (192.168.4.1)
- IP im Browser eintippen und Meldung im Browser überprüfen (*RS232 TO WIFI (SIOS*))

- RFO-BASIC (Oli-BASIC) starten und die Messroutine erstellen und ausführen, Messwerte in die Zwischenablage übernehmen
- Messwerte im Excel für Android über die Zwischenablage einfügen und grafisch darstellen

Das Ergebnis könnte wie folgt aussehen, wenn eine grüne LED mit 1k Vorwiderstand als Messobjekt vorlag.

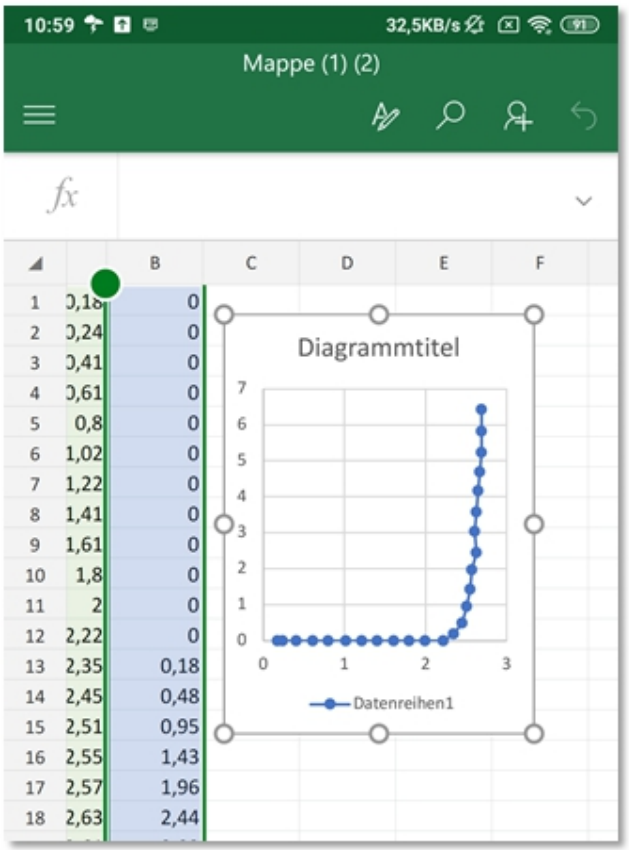


Abbildung 4-59: Dioden Kennlinie via WiFi/RS232 in Mobile-Excel

Das hier nicht weiter erläuterte RFO-Programm ist auf zwei Dateien aufgeteilt. Die *Include* -Datei *esp.bas* enthält die http-Anfragen für den im ESP laufenden http-Server, die eigentliche Messroutine ist dadurch übersichtlicher. Das RFO-Listing *esp.bas* befindet sich im Anhang 5.3.13 und enthält mehr Routinen, als für diese Messung erforderlich.

Die Kennlinienaufnahme erfolgt durch die Steuerspannung in 8-Bit-Auflösung an dem echten Analogausgang 1 (AOUT), die die Messschaltung aus LED und Widerstand als Reihenschaltung speist, die Messung der Gesamtspannung zur Kontrolle von AOUT mit AIN an Analogeingang 1 und der

Spannung am Widerstand gegen Masse mit Analogeingang 2. Über diese Spannung am 1000 Ohm-Widerstand errechnet sich der Dioden Strom in mA. Die Differenz der beiden Spannungen entspricht der Spannung am Messobjekt.

```
!RFO-BASIC
INCLUDE esp.bas
start:
cls:?“U/V    I/mA  des Prüfobjekts”
a$=“Interface: “+int$(hard())
a$=a$+” WiFi-Kennlinienschreiber”
aout(1,112)
DIALOG.MESSAGE a$,url$()+”              Kennlinie abfahren?” , go, “JA”, “NEIN”,“Möglicherweise”
if go = 3 then goto start
IF go <> 1 THEN END
u=0:r=330:c$=””
DO
 aout(1,u*51): CALL dout(u*51)
 u1=ain(1)/51
 u2=ain(2)/51
 i=u2/r*1000
 ux=u1-u2
 a$= USING$(””,”%1.2f%s   %2.2f”,ux,CHR$(9),i)
 PRINT a$
 c$=c$+a$+CHR$(13)+CHR$(10)
 u=u+0.2
 PAUSE 333
UNTIL (ud>3.3) | (i>10) | (u>5)
CLIPBOARD.PUT c$
aout(1,112)
DIALOG.MESSAGE “Messung beendet” ,“Die Messwerte befinden sich auch in der Zwischenablage, um sie in Excel & Co zu verwenden.”, go, “OK”,“Nochmal”
if go=2 then goto start
END
```

Das obige Listing ist in RFO-BASIC und für mobile Android Geräte verfasst.

4.10.2 RFO-BASIC MIT I2C ADC UND DAC

Mehrkanalmessungen mit hoher Auflösung erlaubt der in Abschnitt 4.6 ausführlich dargestellte Analog-Digital-Wandler. Die dort benutzten und erläuterten Mess- und Ansteuerungsroutinen lassen sich auf das hier zu lösende Messproblem übertragen. Hinzu kommt der in Abschnitt 4.7 *12-Bit-Digital-Analog-Wandler - MCP4725* erstmals eingesetzte DAC. Beide Zusatzmodule benutzen I2C, kommen also mit denselben beiden Verbindungsleitungen aus. Die jeweilige Adresse - hier *96* oder *72* - legt fest, welcher der beiden Bausteine angesprochen wird.

Beide Module benutzen im Folgenden GPIO 0 und GPIO 2 am ESP als I2C-Anschlüsse, die bei manchen Boards auch mit D3 und D4 bezeichnet sind. Diese Konstanten sind in ESPBASIC vorgegeben und lassen sich direkt benutzen. Masse und 5 Volt sind ebenfalls parallel verbunden. Die verwendete Messschaltung entspricht überwiegend der Reihenschaltung aus Messobjekt und Messwiderstand in Abbildung 4-46. Gemessen werden die ausgegebene Spannung an die Gesamtschaltung, die vom DAC ausgegeben wird und der Spannungsabfall am Messwiderstand, daraus berechnet sich die Messobjektspannung durch Subtraktion. Der Objektstrom berechnet sich dann aus der Spannung an R_M und dessen Widerstandswert, der im RFO-Anwender-Listing angegeben ist.

Die Kommunikation mit dem ESP8266 erfolgt mittels eines http-Servers, der in ESPBASIC verfasst ist. Diese Zeilen sprechen mit den beiden Hardwareerweiterungen mit drei wesentlichen Anweisungen und vermitteln so zwischen Anwender und Schaltung. Die Anweisungen sind:

- AIN1 - Gesamtspannung
- AIN2 - Spannung am Widerstand
- AOUT1 - Gesamtspannung oder steuerbare Spannungsquelle

Der Aufbau des http-Server Programms entspricht in der Grundstruktur den Ausführungen in Abschnitt 2.6, ergänzt

mit den speziellen I2C-Routinen der verwendeten Hardwaremodule. Die zusätzlichen Anweisungen LED1, LED0 und LDR dienen nur dem Debugging.

```
memclear
Print “DAC & ADC over WiFi”
i2c.setup(0,2)
u = 1.2345678
msgbranch [br]
wait
[br]
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “88”
if cm == “AIN1” then
 Gosub [ain1]
 MyReturnMsg = str(u)
end if
if cm == “AIN2” then
 Gosub [ain2]
 MyReturnMsg = str(u)
end if
if cm == “AOUT1” then
 dac = va
 Gosub [aout]
 MyReturnMsg = str(va)
end if
If cm == “LED1” then Io(po,15,1)
If cm == “LED0” then Io(po,15,0)
if cm == “LDR” then
 MyReturnMsg = str(io(ai))
end if
msgreturn MyReturnMsg
wait
[aout]
i2c.begin(96)
i2c.write(64) ‘DAC 0x40
```

```
i2c.write(dac / 16)
i2c.write(dac % 16 << 4)
i2c.end()
return
[ain1]
MSB = “C1”
gosub [config]
gosub [convert]
return
[ain2]
MSB = “D1”
gosub [config]
gosub [convert]
return
[config]
i2c.begin(72)
i2c.write(1)
i2c.write(hextoint(MSB)) ‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
return
[convert]
vpb = 0.0001875
i2c.begin(72)
i2c.write(0)
i2c.end()
i2c.requestfrom(72,2)
msb = i2c.read()
lsb = i2c.read()
bits = msb * 256 + lsb
if bits > 32767 then bits = bits - 65536
u = bits * vpb
return
```

Die Anwendersoftware ist in diesem Fall ein RFO-Programm, welches zu Beginn die speziellen http-Routinen aus einer separaten Datei mit der *INCLUDE* -Anweisung einbindet.

```
INCLUDE esp.bas
!RFO-BASIC: DAC MCP4725 vs ADS1115
start:
a$="WiFi-Kennlinienschreiber"
DIALOG.MESSAGE a$,url$()+"                Kennlinie abfahren?" , go, "JA", "NEIN","Möglicherweise"
if go = 3 then goto start
IF go <> 1 THEN END
found = 0
?"U/V    I/mA"
dac = O: rm = 988 : c$=""
DO
 aout(1,dac)
 u1=ain(1): u2=ain(2)
 i=u2/rm*1000: ux=u1-u2
 a$= USING$("","%1.2f%s   %2.2f",ux,CHR$(9),i)
 PRINT a$
 c$=c$+a$+CHR$(13)+CHR$(10)
 dac = dac + 32
UNTIL (u1>3.3) | (i>10) | (dac>4000)
CLIPBOARD.PUT c$
aout(1,2000)
DIALOG.MESSAGE "Messung beendet" ,"Die Messwerte befinden sich auch in der Zwischenablage, um sie in Excel & Co zu verwenden.", go, "OK","Nochmal"
if go=2 then goto start
END
```

Die eigentliche Messschleife ist in einer *Do Loop Until* Struktur untergebracht, deren Abbruchbedingung das Überschreiten von Messspannung, Messstrom oder Wandler Wert ist. In der Schleife erfolgen neben der eigentlichen Messung die Formatierung und Vorbereitung von Zeichenketten für die Bildschirmausgabe, sowie für die Zwischenablage.

Am Ende stehen die Messdaten dann formatiert in der Zwischenablage bereit, um z. B. in einer mobilen Excel-Version eingefügt und dargestellt zu werden. Das Ergebnis

entspricht dem in Abbildung 4-59: Dioden Kennlinie via WiFi/RS232 in Mobile-Excel.

Zum Schluss die benutzten RFO-BASIC Definitionen der speziellen http-Routinen in der Datei ESP.BAS:

```
!RFO-BASIC
FN.DEF url$()
FN.RTN “http://192.168.4.1/”
FN.END
FN.DEF AIN(B)
 cmd$=“msg?cmd=ain”+INT$(b)
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF AOUT(B,C)
 cmd$=“msg?cmd=aout”+INT$(b)+”&val=”+STR$(c)
 GRABURL r$, url$()+cmd$
 !r$=“12”
 FN.RTN VAL(r$)
FN.END
```

4.10.3 RFO-BASIC mit I2C DAC und eigenem 10-Bit Eingang

Der LDR des Witty-Cloud Moduls am einzigen Analogeingang des ESP8266 hat viele Anwendungen, stört jedoch eigene Messungen. Aus diesem Grund wir hier ein D1 Mini Modul eingesetzt, da dort der 10-Bit Analogeingang A0 unbeschaltet verfügbar ist. Als steuerbare Spannungsquelle kommt der bereits in Kapitel 4.7 eingesetzte Digital-Analog-Wandler MPC mit 12-Bit Auflösung, also mit 4096 Stufen, zum Einsatz.

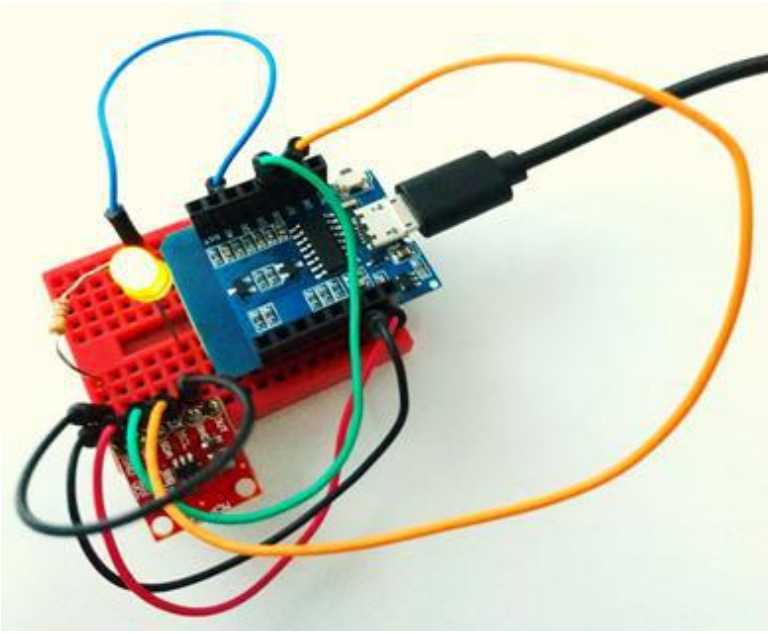


Abbildung 4-60: Kennlinienaufnahme mit D1 Mini und DAC MCP4725 über I2C

Die verwendete Messschaltung entspricht überwiegend Abbildung 4-46, wobei lediglich in diesem einfachen Aufbau darauf verzichtet wird die Analogspannung der gesamten Schaltung zu messen, da nur ein Eingang verfügbar ist. Vielmehr wird der mit Hilfe des Analogeingangs der Analogausgang einmalig kalibriert, so dass diese Spannung durch die Steuerung der Ausgangsspannung bekannt ist.

Zunächst soll die Hardware aufgebaut und überprüft werden. Die I2C-Adresse ist bereits aus dem obigen Kapitel bekannt oder wird mit der Scann-Routine aus 3.8.1 nochmals überprüft. Aus aufbautechnischen Gründen benutzt dieser Versuch die Anschlüsse D7 und D6 als I2C-Bus. Diese Konstanten sind in ESPBASIC schon voreingestellt, was ein Nachschlagen erfreulicher Weise erübrigt. Die exakte Verschaltung von SDA und SCL ist im Listing des http-Servers enthalten. Als Betriebsspannung wird diesmal 5 Volt Vcc vom D1 Mini angelegt. Eine gelbe LED mit 330 Ohm Messwiderstand R_M sollte bei der Inbetriebnahme etwa mit halber Helligkeit leuchten, da der DAC beim Einschalten die halbe Ausgangsspannung liefert.

Nach Fertigstellung des Schaltungsaufbaus muss nun im ESP ein http-Server gestartet werden, der zwischen Smartphone

und Messschaltung über WiFi die Steuerung übernimmt. Es werden nur zwei Aufrufe benötigt:

- Analogausgabe (AOUT)
- Analogeingabe (LDR)

Beide Aufrufe benutzen Digitalwerte, also keine physikalischen Größen. Der Server hat den schon öfter gezeigten Aufbau und könnte wie folgt aussehen:

```
memclear
sda = D6
scl = D7
i2c.setup(sda,scl)
Print “DAC & AI over WiFi”
msgbranch [br]
wait
[br]
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “88”
if cm == “AOUT1” then
 dac = va
 Gosub [aout]
 MyReturnMsg = str(va)
end if
if cm == “LDR” then
 MyReturnMsg = str(io(ai))
end if
msgreturn MyReturnMsg
wait
[aout]
i2c.begin(96)
i2c.write(64) ‘DAC 0x40
i2c.write(dac / 16)
i2c.write(dac % 16 << 4)
i2c.end()
return
```

Nach der I2C-Initialisierung und Festlegung des *msg-Branch* , meldet sich die Routine in einem Browser einmalig mit *DAC & AI over WiFi.* Der über die Adresse 96 angesprochene Wandler wird vom Unterprogramm *[aout]* gesteuert, welches wiederum den Wert *dac* über den Parameter *va* übergeben bekommt mit dem Kommando *AOUT1* . An anderer Stelle wird gezeigt, wie dieser Server auch über die Adresszeile des Browsers angesprochen werden kann. Hier soll die Ansteuerung über Android und RFO/OLI-BASIC erfolgen. Die Abfrage des Analogwerts am Eingang A0 erfolgt unverändert mit dem Kommando LDR.

Zunächst folgen die in einer Datei *esp.bas* ausgelagerten speziellen RFO-Funktionen des Servers, die das Hauptprogramm mit einer *Include* -Anweisung einbindet. Hier ist die URL fest abgelegt!

```
!RFO-BASIC INCLUDE DATEI ZUM KENNLINIENSCHREIBER ÜBER WIFI
FN.DEF url$()
FN.RTN " http://192.168.178.42 /"
FN.END
FN.DEF AOUT(B,C)
 cmd$="msg?cmd=aout"+INT$(b)+"&val="+STR$(c)
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF ip$()
 a$=MID$(url$(),8)
 FN.RTN REPLACE$(a$,"/","")
FN.END
FN.DEF Ldr()
 cmd$="msg?cmd=ldr"
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
!aout(1,0)
!print ldr()
```

Mit den durch Ausrufezeichen auskommentierten Zeilen am Ende dienen der Schaltungsüberprüfung. Mit *aout(1,0)* sollte die LED in der Messschaltung 0 Volt (dunkel) erhalten, mit *print ldr()* sollte ein entsprechender Wert nahe 0 vom Analogeingang erscheinen.

Die eigentliche Kennlinienroutine in RFO-BASIC entspricht dem Listing des vorigen Abschnitts mit dem RS232-Interface. Lediglich die innere Messschleife ist geringfügig anders aufgebaut, da hier nur ein Analogeingang benutzt werden kann. Identische Abschnitte sind mit Punkten markiert und sollten bei Bedarf von oben übernommen werden. Das RFO-BASIC-Programm zur Kennlinienaufnahme einer LED:

```
INCLUDE esp.bas
!DAC MCP4725 vs AI
start:
a$="WiFi-Kennlinienschreiber"
…
dac=0:rm=330:c$=""
DO
 aout(1,dac):pause 50
 u1=dac/4095*5
 u2=ldr()/297 % experimental
 i=u2/rm*1000 % mA
 ux=u1-u2
 a$= USING$("","%1.2f%s   %2.2f",ux,CHR$(9),i)
 PRINT a$
 c$=c$+a$+CHR$(13)+CHR$(10)
 dac = dac + 64
UNTIL (ux>3.3) | (i>6) | (dac>4000)
…
```

Nach der Ansteuerung des DAC sollen mindestens 50 ms dafür sorgen, dass sich der Analogwert eingestellt, der in u1 in eine Spannung umgerechnet wird. Die Messung am Widerstand R_M erfolgt mit u2 = ldr() / 297, wobei der Wert 297 experimentell ist. Diese Ermittlung erfolgte mit der Messschleife, wobei die Ausgabe nicht i und ux, sondern u1

und u2 war und das bei direkter Verbindung zwischen Analogeingang und Analogausgang der Schaltung.

Das Ergebnis im mobilen Excel am Android Smartphone unterscheidet sich nicht wesentlich von Abbildung 4-59 im vorigen Abschnitt, kommt aber mit wesentlich weniger Hardware aus.

4.11 Frequenzgenerator AD9833 über „BitBang“

Ein kleiner Funktionsgenerator mit regelbarem Ausgang ist auf der Platine des AD9833 untergebracht. Die Messungen aus Abschnitt 4.8.1 könnten auch mit dieser Erweiterung erfolgen. Im Gegensatz zum AD9850 stehen drei Signalformen zur Verfügung: Rechteck, Dreieck und Sägezahn.

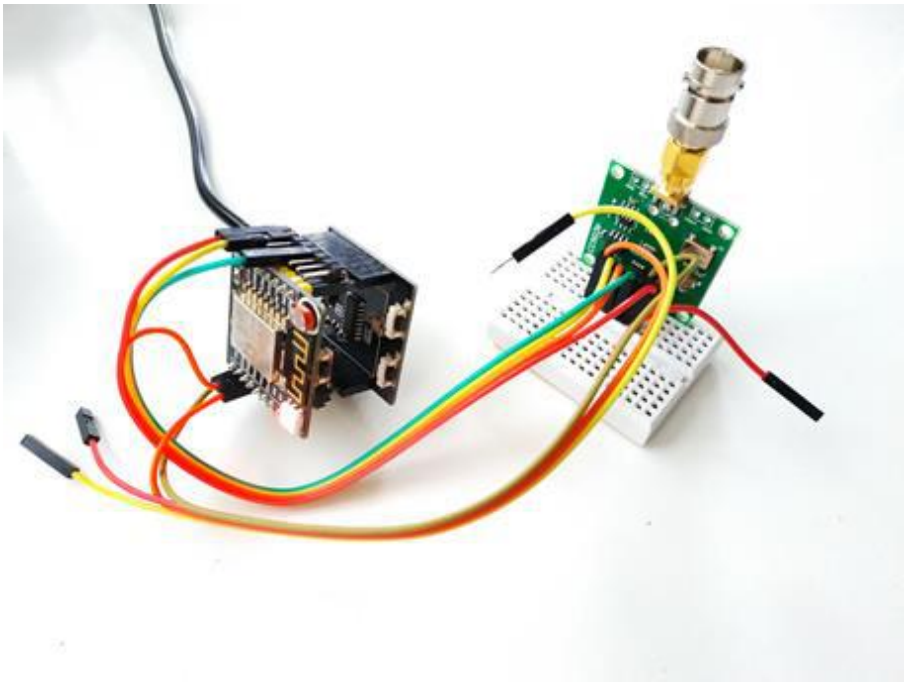


Abbildung 4-61: Witty Cloud und Funktionsgenerator AD9833

Auf der Platine befindet sich zusätzlich ein digitales Potentiometer MCP 41010, mit dem die Ausgangsspannung in 256 Stufen steuerbar ist. Die Ansteuerung dieses Potentiometers über SPI ist als Beispiel im Abschnitt 3.9 aufgeführt und ist im folgenden Abschnitt nicht mehr enthalten. Dort wird ebenfalls auf ein Problem bei der SPI-Ansteuerung unter ESP8266BASIC 3.0 Branch 69 eingegangen, wonach die SPI-Modi 2 und 3 nicht korrekt funktionieren. Aus diesem Grund gelingt die Ansteuerung des Funktionsgenerators AD9833 nur mit eigenen Routinen, die in BASIC verfasst sind und darum entsprechend langsam reagieren. Um einen solchen Baustein überhaupt unter ESPBASIC zu benutzen, folgt hier das Listing zur

Ansteuerung und der SPI-Leitungen. Dafür ist es erforderlich die einzelnen Bits einzeln über die Datenleitung zu senden und auch den Sendetakt einzeln zu schalten. Dieses „Leitungsgeklapper“ findet man im englischen Sprachraum unter „BitBang“. Die Details zum Listing folgen danach.

```
memclear
CSM = 2 ‘CS MCP41010
FSY = 16 ‘CS AD9833
DAT = 13 ‘HARD MOSI (12 is MISO)
CLK = 14 ‘HARD SPI
data = 8448
gosub [bang]
data = 20679
gosub [bang]
data = 16384
gosub [bang]
data = 49152
gosub [bang]
data = 8192
gosub [bang]
freq = 1000000
textbox freq
button “Frequenz”,[setf]
slider Vu,0,255
button “Vu”,[setg]
wait
[setf]
gosub [set]
wait
[setg]
gosub [gain]
wait
[gain] ‘MCP41010
CS = CSM
data = 4352 or (Vu and 255)
gosub [bang]
```

```
return
[set] ‘freq
FreqReg = (freq * pow(2, 28)) / 25000000
CS = FSY
MSB = FreqReg and 268419072
MSB = MSB >> 14
LSB = FreqReg and 16383
LSB = LSB or 16384
MSB = MSB or 16384
data = 8448
gosub [bang]
data = LSB
gosub [bang]
data = MSB
gosub [bang]
data = 8192 ‘Sinus
‘data = 8194 ‘Dreieck
‘data = 8224 ‘Rechteck
gosub [bang]
return
[bang] ‘data
pointer = 32768
io(po,CS,0)
for i = 0 to 15
 bit = (data and pointer) > 0
 io(po,DAT,bit)
 io(po,CLK,0)
 io(po,CLK,1)
 pointer = pointer >> 1
next i
io(po,CS,1)
return
```

Zu Beginn erfolgt die Festlegung der verschiedenen Verbindungen oder Anschlüsse. Die CS-Leitung des Digitalpotentiometers ist demnach mit GPIO 02 verbunden, die des Generators mit GPIO 16. Anhand entsprechender

Ausführungen im Datenblatt des AD9833 folgt eine Initialisierung, dabei erfolgt die Datenübertragung im Unterprogramm *[bang]* . Im Browser sorgt ein *Slider* für die Spannungssteuerung im Bereich 0 bis 255 und eine *Textbox* dient der Anzeige und Eingabe der Frequenz, die sich mit dem *Button* einstellen lässt.

Die einzelnen 16 Bit der Variablen *data* überträgt die Routine bang im Gänsemarsch über die Leitung DAT. Jedes Bit erfordert einen 0/1-Wechsel (Klappern) an der Leitung CLK als Takt. Auf diese Art und Weise kann man auch die SPI-Modi 2 und 3 umsetzten, allerdings in Zeitlupe.

4.12 Rheinturmuhr in Neopixel am Internet

Der Rheinturm in Düsseldorf zeigt die Uhrzeit mit einer sogenannten Dezimaluhr an. Dabei werden 39 Lampen so gesteuert, dass die Einer- und Zehnerstellen von Stunde, Minute und Sekunde getrennt leuchten. Die Einerstellen bestehen dadurch aus neun Lampen – bei einer Null sind die neun Lampen aus. Die Zehnerstellen benutzen für die Sekunden und Minuten jeweils fünf -, die Stunden benötigen nur zwei Lampen. Auf den Seiten von *hjberndt.de* gibt es verschiedene Varianten dieser Rheinturmuhr, die alle ein gemeinsames Problem aufweisen: die Synchronisation. Die genaue Uhrzeit wird heute bei PC und Smartphone über das Internet bezogen und damit bietet sich dieser Weg jetzt auch für eigene Konstruktionen an. Ein ESP8266 bietet sich dafür an. In [3] ist es ein Digispark, der in C/C++ diese Anzeige steuert, hier wird die ESPBASIC-Variante in wenigen Zeilen realisiert, Synchronisation inklusive.

ESPBASIC in der 2MB-Version enthält die in Kapitel 3.4 beschriebe NeoPixel-Bibliothek, die in diesem Beispiel benutzt werden soll. Mit ganzen 6 Routinen wird der Farbstreifen unterstützt. Hier werden nur die Befehle *neo.setup* , *neo* , *neo.cls* und *neo.stripcolor* benötigt. Die Zeit-Routinen kommen hier ebenfalls zum Einsatz, ebenso ein Timer. Diese Uhr wird durch die in ESPBASIC intern eingebauten Bibliotheken automatisch über das Internet gestellt, wenn der ESP8266 per Router Zugang zum Netz hat. Insgesamt ergibt sich ein erfreulich kurzes Gesamt-Listing, wenn man bedenkt, dass die komplette Steuerung und Synchronisation für die sogenannte Dezimaluhr darin enthalten ist.

4.12.1 Sekundentakt

Die Grundfunktion der Uhr an Pin *0* lässt sich anhand der kurzen Sekundenanzeige am übersichtlichsten darstellen:

```
Neo.setup(0)
Neo.cls()
```

```
Neo(00,20,0,0)
Neo(10,20,0,0)
timer 1000,[uhr]
Wait
[uhr]
t$ = time(“hour:min:sec”)
s0 = val(mid(t$,8,1))
if s0 = 0 then
 neo.stripcolor(1,9,1,1,1)
 s1 = val(mid(t$,7,1))
print s1
 neo.stripcolor(11,10+s1,10,10,10)
 if s1 = 0 then
  neo.stripcolor(11,10+5,1,1,1)
 endif
endif
neo.stripcolor(1,s0,10,10,10)
Wait
```

Die Steuerung erfolgt über einen Timer. Dabei wird ein ESPBASIC-“Branch” im Timer-Intervall - 1000 ms – aufgerufen und die jeweilige Sekunde aktualisiert zur Anzeige gebracht. Im Listing ist das der Abschnitt *[uhr]* . Davor erfolgt die Initialisierung duch Festlegung des benutzten Data-Pins mit *neo.setup(0)* , löschen aller LED mit *neo.cls* und schließlich das diskrete Einschalten der roten Markierungen zwischen den verschiedenen LED-Abschnitten.

In der Timer-Routine [uhr] erfolgt die Zeitabfrage in *t$* in der Form “hh:mm:ss” duch Übergabe eines entsprechenden Formatierungs-Strings. Mit dem Wert in *s0* werden die Einerstellen der Sekunden gesteuert. Die Sekunden erhält man als numerischen Wert mit Hilfe der BASIC Funktionen *val* und *mid* ab der siebten Stelle der Zeichenkette. Mit *neo.stripcolor* werden ganze Bereiche von LED mit den angegebenen RGB-Werten geschaltet. Entspricht die Einerstelle in *s0* dem Wert 0, dann erfolgt ein Dunkelschalten *neo.stripcolor(1,9,1,1,1)* der LED 1 bis 9 und die Zehnerstelle wird entsprechend isoliert und aktualisiert angezeigt. Mittels

neo.stripcolor(1,s0,10,10,10) erfolgt die Darstellung der Einerstelle in diesem kurzen Beispiel am Ende durch Einschalten der LED 1 bis *s0* mit geringer Helligkeit – und somit geringem Stromverbrauch - für alle drei Farben.

4.12.2 VOLLZEIT

Im Gesamt-Listing folgen dann auch alle anderen Aktualisierungen, die im Prinzip genau so funktionieren, nur mit anderen Variablen und LED-Nummern. Es gibt ja jeweils 9, 5, 9, 5, 9, 2 LED-Abschnitte für Sekunden, Minuten und Stunden in Dezimalschreibweise, wie oben schon erwähnt. Da z.B. ein Witty Cloud Modul auch über einen LDR verfügt, wird mit *io(ai)* die Helligkeit *i* der Anzeige berücksichtigt. Das kleine *o* legt die Helligkeit für "aus" fest - nicht zu verwechseln mit einer 0. Am Ende folgt noch der Lichterwechsel der LED an der Turmspitze - hier in grün - ein Wechselblinken im Sekundentakt. Die erste Anweisung sorgt dafür, dass der ESP sowohl als eigener AP, als auch als Station erreichbar ist.

```
'wifiapsta
Neo.setup(0)
Neo.cls()
Neo(00,20,00,00)
Neo(10,20,00,00)
Neo(16,20,10,00)
Neo(26,20,10,00)
Neo(32,20,00,00)
Neo(42,20,00,00)
print "RTC WS PIN 0"
timer 1000,[uhr]
Wait
[uhr]
i = io(ai)/15+5
o = 1
t$ = time("hour:min:sec")
'Print t$
s0 = val(mid(t$,8,1))
if s0 = 0 then
```

```
 neo.stripcolor(1,9,0,0,o)
 s1 = val(mid(t$,7,1))
 neo.stripcolor(11,10+s1,i,i,i)
 if s1 = 0 then neo.stripcolor(11,10+5,0,0,o)
 m0 = val(mid(t$,5,1))
 neo.stripcolor(17,16+m0,i,i,i)
 if m0 = 0 then neo.stripcolor(17,16+9,0,0,o)
 m1 = val(mid(t$,4,1))
 neo.stripcolor(27,26+m1,i,i,i)
 if m1 = 0 then neo.stripcolor(27,26+5,0,0,o)
 h0 = val(mid(t$,2,1))
 neo.stripcolor(33,32+h0,i,i,i)
 if h0 = 0 then neo.stripcolor(33,32+9,0,0,o)
 h1 = val(mid(t$,1,1))
 neo.stripcolor(43,42+h1,i,i,i)
 if h1 = 0 then neo.stripcolor(43,42+2,0,0,o)
endif
neo.stripcolor(1,s0,i,i,i)
b = s0 and 1
Neo(45,0,b*20,00)
Neo(46,0,(1-b)*20,00)
Memclear
Wait
```

4.12.3 Synchronisierung und Reset

Ist die Uhr als voreingestelltes Autostart-Programm *\default.bas* abgespeichert, so startet ESPBASIC sie automatisch nach einem Reset oder Absturz. Im Detail gilt dann der folgende Ablauf:

ESP8266 bootet und ESPBASIC übernimmt, indem es gestartet wird. Dieses versucht dann eine Verbindung mit einem im Setup gespeicherten Access Point (Router) herzustellen. Schägt der Versuch innerhalb einer gewissen Zeit fehl, schaltet ESPBASIC den eigenen Access-Point an, so dass

ein Browser über die IP 192.168.4.1 im lokalen „ESP…“-Netzwerk Zugriff bekommt. Bei Misserfolg, Start des eigenen AP mit Namen “ESP…”. Nach 30 Sekunden startet *\default.bas* , falls dass so in den Settings abgehakt ist. Im Notfall kann innerhalb dieser 30 Sekunden der Autostart noch unterbunden werden (z. B. durch SAVE oder Änderung der Settings). Die Uhr des ESP8266 steht auf “00:00:00”. Das bedeutet, dass ohne Internetanbindung keine Änderung eintritt und die Uhr dunkel bleibt - von den Positionsleuchten abgesehen. Ist aber z.B. eine FritzBox mit Zugangspasswort in den Settings eingetragen und das Hinzufügen von Geräten in den Einstellungen des Routers erlaubt, verbindet sich der ESP damit und ist unter 192.168.178.xxx erreichbar. Dann gestaltet sich ein erster Start wie folgt:

Der ESP wird mit Spannung versorgt und startet das BASIC-System, welches nun die Verbindung mit dem Router herstellt - die blaue LED leuchtet kurz auf. Nach weiteren 30 Sekunden startet das obige Listing und die Positionsleuchten gehen an. Die BASIC-interne Bibliothek versucht nun selber die Zeit über einen Zeit-Server im Internet zu synchronisieren. Nach einer Weile - oder auch sofort - laufen die unteren Sekunden los, um nach erreichen der 10. Sekunde, die gesamte Uhrzeit in der Rheinturmuhr üblichen Dezimalform anzuzeigen. Bleibt die Uhr stationär, kann auf den eigenen AP verzichtet werden und die Uhr sollte immer genau gehen, solange das Internet ab und zu die Zeit liefern kann. Zur Entwurfszeit kann es sinnvoll sein, mit *wifiapsta* immer beide Zugänge für den Browser offen zu halten.

Anmerkung: Für den mobilen Einsatz kann die Uhr auch mit einem Smartphone und gelegentlichem Tethering (persönlicher Hotspot) synchronisiert werden. Aktuelle Smartphones erlauben Wlan und Hotspot gleichzeitig – sogar ohne SIM. Zwischendurch läuft die Uhr mit geringer Gangungenauigkeit allein weiter bis der Hotspot wieder aktiviert ist, der ESP sich nach einer Weile wieder verbindet und die Internetzeit holt.

4.12.4 Erweiterung mit Schall und Licht

Die Ganggenauigkeit von Uhren konnte und kann bei FM/UKW-Sendern, die nicht über Umwege/Digitalisierung

verzögert sind, beobachtet oder gehört werden. Auch Radiosender über Satellit sind meist nur gering verzögert. Allerdings scheinen die terrestrischen Ausstrahlungen auch über diesen Weg versorgt zu werden, wodurch als Referenz nur noch DCF77 als Langwellen Zeitgeber zuverlässig ist. Auch Funkuhren synchronisieren nicht im Sekundentakt. Bereits eine GPS-Variante benutzt Pips mit Hilfe eines Piezo-Beepers. Die ESP8266-Uhr mit kann in dieser Hinsicht wie folgt erweitert werden:

- fünf Sekunden vor der vollen Stunde blinken die oberen RGB-LED des Streifens hell
- fünf Sekunden vor der vollen Stunde gibt es Pips.
- zur vollen Stunde gibt es einen langen Pip
- zur halben Stunde gibt es einen kurzen Pip

Die Pips werden über PWM realisiert. Da ESPBASIC die PWM-Frequenz mit *pwmfreq* (vgl. 2.3.4) einstellen kann, wird diese auf 1000 Hz gestellt. Bei Quasi-Analogausgaben über Puls-Breiten-Modulation (PWM) entsteht bei einem Ausgabewert von 127 ein symmetrisches Rechtecksignal mit einem Mittelwert der halben Amplitude. Wird ein Piezo-Beeper an GPIO 15 angeschlossen, so kann man dieses Signal hören. Das Signal bleibt erhalten, auch wenn das Programm weiter läuft. Mit dieser Methode wird am Witty Cloud Pin 15 per PWM angesteuert und damit gleichzeitig die dort angeschlossene interne rote LED mit halber Spannung versorgt. Der kurze Pip ist in einem Unterprogramm ausgelagert und wird entsprechend mit *gosub* aufgerufen. Für die Erweiterung wird *memclear* und *wait* des obigen Listings ersetzt durch:

```
x = val(mid(t$,4,2)) * 100
x = val(mid(t$,7,2)) + x
If (x > 5954) then
 Neo.stripcolor(50,60,100,100,100)
 gosub [pip]
 delay 60
 Neo.stripcolor(50,60,0,0,0)
Endif
```

```
if (x = 3000) then
 gosub [pip]
endif
if x = 0 then
 io(pwo,15,127)
else
 io(pwo,15,0)
endif
Memclear
Wait
[pip]
pwmfreq 1000
io(pwo,15,127)
delay 40
io(pwo,15,0)
return
```

Die Zeile *Neo.stripcolor(50,60,100,100,100)* sorgt dafür, dass zur vollen Stunde die oberen NeoPixel relativ hell aufblinken, so dass auch optisch Aufmerksamkeit eingefordert wird.

Mit dieser Uhr schließt das Buch die Anwendungen von ESPBASIC.

---

5 ANHANG

5.1 EINRICHTUNG/INSTALLATION

Die Einrichtung von ESP8266BASIC besteht hauptsächlich aus der einmaligen Übertragung der Binärdatei in den Speicher eines ESP8266-Bausteins. Das sogenannte „flashen“ erfolgt über den USB-Anschluss und einem entsprechenden Programm. Die Englische Anleitung für PC‘s ist auf ESP8266BASIC.com zu finden und wird hier kurz aufgeführt. Es ist auch möglich mittels USB-Host-Adapter die Binärdateien direkt vom Android-Smartphone aus zu übertragen.

5.1.1 WINDOWS

Die vom Autor vorgeschlagene Vorgehensweise findet man unter

https://www.esp8266basic.com/flashing-instructions.html

Weiter unter Downloads auf der Homepage von *ESP8266Basic.com* kann der Quelltext auf GitGub unter ‚*Full Source on GitHub* ‘ angesteuert werden. Im dortigen Verzeichnis

https://github.com/esp8266/Basic/tree/NewWebSockets/Flasher

liegt oder lag das Programm ***ESP_Basic_Flasher.exe*** . Mit der rechten Maustaste und *Link speichern unter* landet die Datei auf dem PC im Download-Verzeichnis. Nach dem Start kann der Bildschirm wie folgt aussehen.

Abbildung 5-1: ESPBASIC-Flasher für Windows

Hier eine Art Kochrezept zur Übertragung.

- ESP8266 über USB mit dem PC verbinden.
- COM-Schnittstelle im Flash-Programm auswählen.
- Connect-Button betätigen, rotes Feld wird bei Erfolg grün.
- Flash-Größe auswählen (2MB als Beispiel für ESP8266F).
- Firmware Flash drücken und abwarten.

Bei Erfolg meldet sich ESPBASIC im großen Terminalfenster des Flash-Programms. In der oberen Zeile könnten sogar Zeichen an die serielle Schnittstelle des ESP gesendet werden, wie in Abschnitt 2.1 beschrieben. Der PC ist nun nicht mehr erforderlich, da ESPBASIC ab sofort auch über jeden Browser erreichbar ist, sobald der ESP8266 mit Spannung versorgt ist.

Nach einem Reset sollte sich ESPBASIC als eigener Hotspot mit einer Kennung nach dem Muster ‚ESP:xx.xx.xx.xx.xx' im WiFi zeigen. Verbindet man sich mit diesem Netzwerk, so ist darüber zunächst kein Internet mehr verfügbar, jedoch ESPBASIC im Browser unter der URL bzw. IP **http://192.168.4.1**

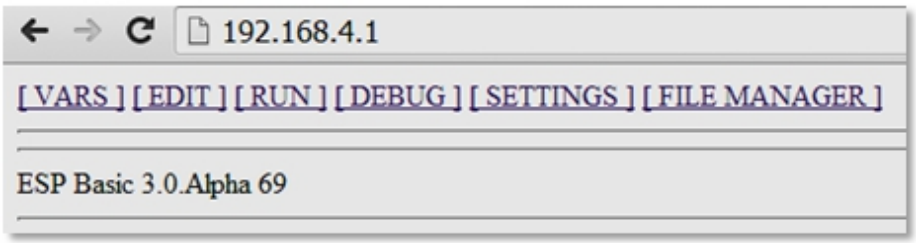


Abbildung 5-2: Erste Kontaktaufnahme im Browser

Der Programmierung nach Kapitel 1 steht nichts mehr im Weg und mit der BASIC-Zeile *Print 88* und anschließendem *Save* und *Run* zeigt der Browser die Zahl 88 als ersten Test.

Unter Settings sollte nun die Verbindung über den heimischen Router erfolgen, durch entsprechende Angaben von Name und Passwort. Nach einem Save und Restart ist dann ESPBASIC unter einer vom Router vergebenen IP zu erreichen und PC

und ESP können wieder auf das Internet zugreifen. Die IP erfährt man im Flash-Programm, wenn der ESP über USB noch verbunden ist, oder jedem anderen seriellen USB-Terminal, da ESPBASIC über die eingebaute serielle Schnittstelle und den Leitungen TX/RX kommuniziert (vgl. Kapitel 2).

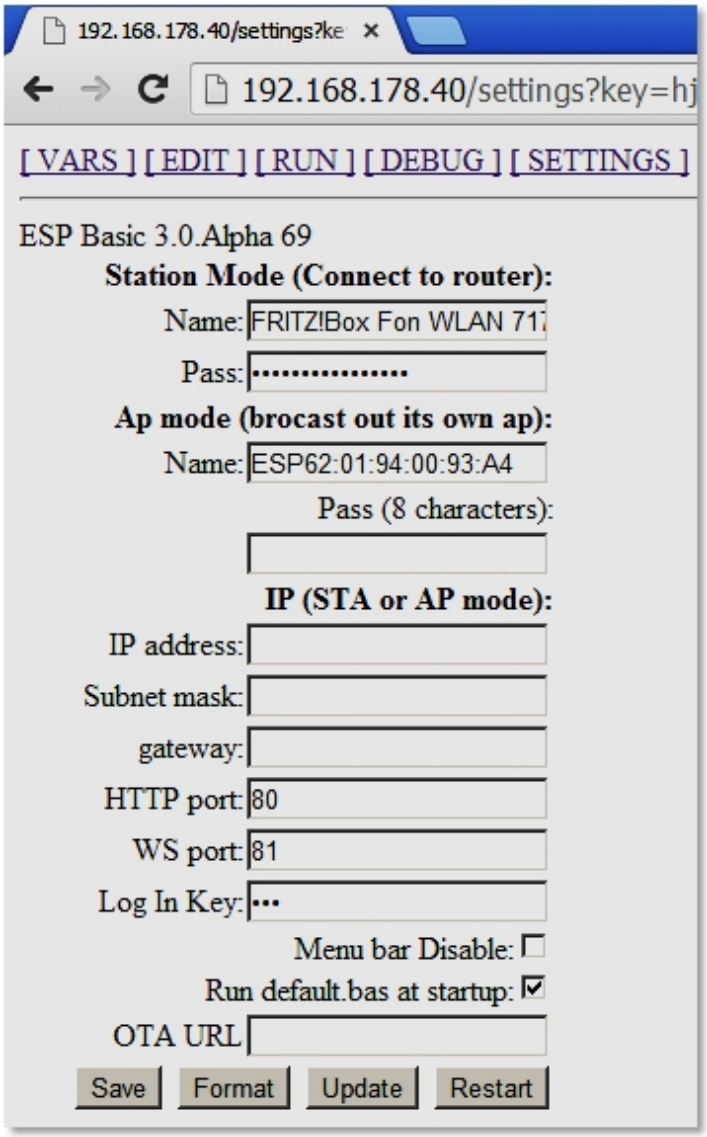


Abbildung 5-3: Eintragungen für den Stationsbetrieb in den Einstellungen

5.1.2 ANDROID

Mit einem HOST-Adapterkabel kann ein Android-Smartphone über USB mit einem ESP8266-Board direkt, wie ein PC, kommunizieren. Dies ist im Abschnitt 2.1 beschrieben. Zur Übertragung von Speicherinhalten ist die App *ESP8266 Loader* aus dem Playstore vorgesehen. Damit lassen sich

Binärdateien mit sehr geringem Aufwand von Android Geräten zum ESP8266 übertragen. Die Binärdateien von ESPBASIC sind oder waren unter *GitHub* zu finden. Der Link dazu steht auf der Homepage von *ESP8266Basic.com* unter *Download* und *Full Source on GitHub* .

Die Binärdateien befinden sich auf *GitHub* dann unter *Basic/Flasher/Build* in weiteren nach Flash Größe benannten Unterverzeichnissen mit dem Namen ***ESP8266Basic.cpp.bin*** . Der Link zur 2MB-Version ist z.B.:

https://github.com/esp8266/Basic/tree/NewWebSockets/Flasher/Build/2M

Alternativ kann das gesamte Zip-Archiv *Basic-NewWebSockets* auf dem Smartphone entpackt - und mit dem *ESP8266 Loader* dann zu den entsprechenden Verzeichnissen navigiert werden.

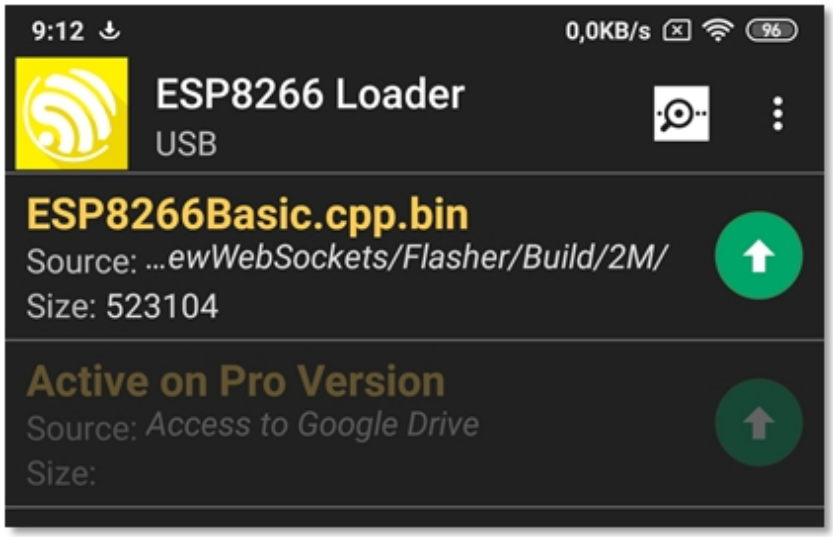


Abbildung 5-4: Flash-App auf Android

Das Tool braucht eventuell mehrere Anlaufversuche, für den Fall, dass der ESP8266-Baustein mit laufenden Anwendungen zu beschäftigt ist, um die Datei erfolgreich zu übertragen. Ein schnelles Blinken der blauen LED ist Erfolg versprechend. In den *Settings* sollte die *Reset Method* entsprechend dem Board eingestellt sein. Für das Witty Cloud Board ist das *nodemcu* , wobei ein Reset und die Übertragung automatisch erfolgt.

Nach der Übertragung kann mit dem eingebauten Terminal (Symbol Lupe oben rechts) dann verfahren werden, wie im

weiter oben beschriebenen Flash-Programm und dem dortigen Terminal, um die ersten Schritte zu gehen.

5.2 Referenz bt93.js

Es folgt eine kurze Referenz der Funktionen, Variablen und Konstanten dieser einfachen Bibliothek. JavaScript untescheidet zwischen Groß- und Kleinschreibung!

Grafik (AN/AUS)

Schaltet die Grafik an oder aus und initialisiert mit AN die nötigen Variablen. AUS wird nicht benutzt.

Diagramm (xa, xe, xs, ya, ye, ys)

Zeichnet eine Diagrammfläche mit Gitter und zwei Achsen. Die Parameter sind für x und y gleich und bedeuten Anfang, Ende und Schrittweite. Ist die Schrittweite 0, so wird eine 10er Teilung benutzt. Bei logarithmischer Achse wird der dritte Achsenparameter ignoriert. Die Zeichenketten *xachse* , *yachse* und *titel* können entsprechende Bezeichner enthalten.

Die Variable *diagrammtyp* ist voreingestellt auf XLINYLIN für ein doppelt-lineares Diagramm. Mit den Konstanten XLINYLIN, XLINYLOG, XLOGYLIN und XLOGYLOG kann der globalen Variable *diagrammtyp* eine Variante zugewiesen werden, bevor Diagramm aufgerufen wird.

DiaPunkt (x, y)

Markiert einen Punkt in Diagrammeinheiten mit dem *marktyp* und der *markgroesse,* wenn vorher *Diagramm* aufgerufen wurde. Die markgroesse (3) ist in Pixel angegeben. Der *marktyp* kann eine Kombination von KREUZ, RAUTE, KASTEN, DELTA, PUNKT und KREIS sein. Diese Konstanten können addiert oder odiert sein. Der Punkt erscheint in der Farbe, die die Variable *farbe* enthält.

DiaLinie (x1, y1, x2, y2)

Zeichnet eine Linie in Diagrammeinheiten und der zuletzt der Variablen *farbe* zugewiesenen Farbe, wenn vorher *Diagramm* aufgerufen wurde.

DiaText (x, y, s)

Schreibt eine Zeichenkette *s* als Konstante oder Variable an den Diagrammpunkt x/y. Für die Farbe gilt das Gleiche wie bei Punkt und Linie.

cls()

Löscht den gesamten Canvas oder die Zeichenfläche mit der zuletzt eingestellten *farbe* .

hintergrund()

Löscht die Diagrammfläche mit der zuletzt eingestellten *farbe* .

printxy(x, y, s)

Wie *DiaText* , aber mit Canvas-Koordinaten (Pixel).

draw(x1, y1, x2, y2, color)

Wie *DiaLinie* , aber mit Canvas-Koorinaten und ohne die Variable *farbe* .

plot(x, y, color)

Setzt ein Pixel in der Farbe color auf die Canvas-Koorinate x/y.

Globale Variablen:

diagrammtyp, marktyp, markgroesse,

xachse, yachse, titel, farbe,

xwork, ywork, wwork, hwork

Die letzten vier Variablen enthalten die Diagrammfläche in Pixeleinheiten.

5.3 LISTINGS

Einige Listings stehen hier, da sie meist etwas länger ausfallen und weiter vorne eher stören würden.

5.3.1 CANVAS UHR ALS ESPBASIC-LISTING

Weiteres Beispiel zu Abschnitt 4.2

```
html |<html><body bgcolor="whitesmoke"><br>|
html |<canvas id="myCanvas" width=640 height=480|
html | style="border:0px solid #c3c3c3"></canvas>|
html |<script src="http://hjberndt.de/soft/bt93.js"|
html | type="text/javascript"></script>|
html |<script type="text/javascript">|
html |function Ziffernblatt()|
html |{var x,xx,i,j;|
html | x=0;|
html | for(i=1;i<=12;i++)|
html | {farbe= WEISS; DiaPunkt(Gsin(x),Gcos(x));|
html |  xx=x;|
html |  for(j=1;j<=4;j++)|
html |  {xx=xx+360/60;|
html |   farbe=GRAU;|
html |   DiaLinie(Gsin(xx),Gcos(xx),0.98*Gsin(xx),0.98*Gcos(xx));|
html |  }|
html |  x=x+360/12;|
html | }|
html |}|
html |function Zeiger(h,m,s,c)|
html |{var a;|
html | switch(c)|
html | {case "m":|
html |   a=(m/60*360);|
html |   farbe=ORANGE;|
html |   DiaLinie(0,0,0.95*Gsin(a),0.95*Gcos(a));|
html |   a=m*360/12/60;|
```

```
html |   a=a+(h/24*2*360);|
html |   farbe=ORANGE;|
html |   DiaLinie(0,0,0.75*Gsin(a),0.75*Gcos(a));|
html |   ctx.lineWidth=3;|
html |   break;|
html |  case “s”: a=(s/60*360);|
html |   farbe=ROT;|
html |   DiaLinie(0,0,0.95*Gsin(a),0.95*Gcos(a));|
html |   break;|
html | }|
html |}|
html |function Uhr()|
html |{var h,m,s,now=new Date();|
html | Grafik(AN);markgroesse=3;ctx.lineCap=“round”;|
html | ctx.lineWidth=3;|
html | hwork=ctx.canvas.height-4*markgroesse; wwork=hwork;|
html | ywork=2*markgroesse;|
html | xwork=ctx.canvas.width / 2 - wwork / 2;|
html | farbe=SCHWARZ; Diagramm(-1,1,0,-1,1,0);|
html | farbe=WEISS;   Ziffernblatt();|
html | h=now.getHours(); m=now.getMinutes(); s=now.getSeconds();|
html | Zeiger(h,m,s,“m”); Zeiger(h,m,s,“s”); |
html |}|
html |window.setInterval(“Uhr()”, 1000)|
html |</script>|
html |</body>|
html |</html>|
wait
```

5.3.2 YT-SCHREIBER MIT JAVASCRIPT

Listing zu Abschnitt 4.2.2

```
ti = “30 s”
td = “100 ms”
meter y,0,1000
idy = htmlid()
dropdown td, “dt, 10 ms, 20 ms, 30 ms, 50 ms, 100 ms, 200 ms, 500 ms, 1000 ms”
```

```
dropdown ti, “MBE, 10 s, 20 s, 30 s, 60 s, 120 s, 180 s, 600 s”
idti = htmlid()
t0 = millis()
textbox text
text = 0
idt = htmlid()
button “t=0”, [reset]
timer 500 ,[messen]
html |<html><body bgcolor=“whitesmoke”><br>|
html |<canvas id=“myCanvas” width=480 height=320 style=“border:0px solid #c3c3c3”></canvas>|
html |<script src=”/file?file=bt93.js” type=“text/javascript”></script><script type=“text/javascript”>|
html |var meter = document.getElementById(“| & htmlvar(idy) & |”);|
html |var tmaxs = document.getElementById(“| & htmlvar(idti) & |”);|
html |var idt = document.getElementById(“| & htmlvar(idt) & |”);|
html |const NMAX=100;var filled=false;var ix=0; var tmax=100;var t,y;|
html |var TabY = new Array(NMAX+1);var TabX = new Array(NMAX+1);|
html |function Messen(){|
html |t=idt.value; y=meter.value;ix = (Math.round(NMAX*t/tmax));|
html |TabY[ix]=y;TabX[ix]=t;}|
html |function Anzeigen(){var i;
Grafik(AN);Messen();yachse=””;xachse=“t/s”;|
html |farbe =“GhostWhite ” ;cls();farbe=“white “;hintergrund();
farbe=“DarkGrey” ; |
html |var res = tmaxs.value.substring(0, 3);tmax=parseInt(res);|
html |var step =
10;if(tmax<100)step=5;      if(tmax<30)step=2;if(tmax<20)step=1;if(tmax>100)s
tep=0;|
html |Diagramm(0,tmax,step,0,1000,0); farbe=“gray”;
DiaLinie(0,500,tmax,500);|
html |farbe=“blue”;DiaText(0,-120,ix);DiaPunkt(t,meter.value);farbe=ROT;|
html |for(i=0;i<NMAX;i++)DiaLinie(TabX[i],TabY[i],TabX[i+1],TabY[i+1]);}|
html |window.setInterval(“Anzeigen()”, 50);</script><br></body></html>|
[reset]
t0 = (millis()/1000)
wait
[messen]
```

```
y = io(ai)
if (text)>val(ti) then t0 = (millis()/1000)
text = (millis()/1000)-t0
timer val(td),[messen]
wait
```

5.3.3 HTTP-SERVER FÜR DAS COMPULAB (RS232)

Routine zu Abschnitt 0 zum Einsatz des älteren CompuLab-Interfaces.

```
memclear
x = -1
y = -1
timer 1000, [tm]
baud = 19200
Print “RS232 TO WIFI (CLAB)”
serial2begin baud, 5, 4 ‘TX/RX
Serial2Print chr(81)
Serial2Print chr(15)‘00001111 Ready-Muster
Serial2branch [rx]
msgbranch [msgbr]
wait
[msgbr]p
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “Server CompuLab”
cls
Print cm&” “&str(va)
If cm == “LED1” then Io(po,2,0)
If cm == “LED0” then Io(po,2,1)
if cm == “LDR” then
 MyReturnMsg = str(io(ai))
end if
if cm == “SEND” then
 Serial2Print chr(va)
 MyReturnMsg = ““‘str(va)
end if
```

```
if cm == “HARD” then
 Serial2Print chr(1)
 MyReturnMsg = ””
end if
if cm = “BAUD” then
 Serial2end
 Serial2begin va, 5, 4
 MyReturnMsg = str(baud)
 baud = va
end if
‘+++++COMPULAB+++++++++++++++++
if cm == “AIN1” then
 Serial2Print chr(48)
 MyReturnMsg = ””
end if
if cm == “TAIN1” then
 Serial2Print chr(60)
 MyReturnMsg = ””
end if
if cm == “AIN2” then
 Serial2Print chr(58)
 MyReturnMsg = ””
end if
if cm == “DIN” then
 Serial2Print chr(211)
 MyReturnMsg = ””
end if
if cm == “DOUT” then
 Serial2Print chr(81)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
‘+++++COMPULAB+++++++++++++++++
If MyReturnMsg <> ”” then msgreturn MyReturnMsg
io(po,12,y)
y = not y
```

```
wait
[rx]
serial2input z$
mi = millis()
m$ = str(mi)
ret = asc(mid(z$,1,1))
if cm == “TAIN1” then ret = m$ & “,” & ret
Msgreturn ret
Return
[tm]
io(pwo,12,(x and 15))
x = not x
wait
```

5.3.4 HTTP-Server für das Sios-Interface (RS232)

Routine zu Abschnitt 0 zum Einsatz des älteren Sios-Interfaces.

```
memclear
x = -1
y = -1
timer 1000, [tm]
baud = 19200
Print “RS232 TO WIFI (SIOS)”
serial2begin baud, 5, 4 ‘TX/RX
Serial2Print chr(81)
Serial2Print chr(15)
Serial2branch [rx]
msgbranch [msgbr]
wait
[msgbr]
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “Server Sios”
cls
‘Print cm&” “&str(va)
```

```
If cm == “LED1” then Io(po,15,1)
If cm == “LED0” then Io(po,15,0)
if cm == “LDR” then
 MyReturnMsg = str(io(ai))
end if
if cm == “SEND” then
 Serial2Print chr(va)
 MyReturnMsg = ””
end if
if cm == “HARD” then
 Serial2Print chr(1)
 MyReturnMsg = ””
end if
if cm = “BAUD” then
 Serial2end
 Serial2begin va, 5, 4
 MyReturnMsg = str(baud)
 baud = va
end if
‘+++++SIOS+++++++++++++++
if cm == “AIN1” then
 Serial2Print chr(48)
 MyReturnMsg = ””
end if
if cm == “TAIN1” then
 Serial2Print chr(48)
 MyReturnMsg = ””
end if
if cm == “AIN2” then
 Serial2Print chr(49)
 MyReturnMsg = ””
end if
if cm == “DIN” then
 Serial2Print chr(32)
 MyReturnMsg = ””
end if
```

```
if cm == “DOUT” then
 Serial2Print chr(16)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
if cm == “AOUT1” then
 Serial2Print chr(64)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
‘+++++SIOS++++++++++++++++
If MyReturnMsg <> ”” then msgreturn MyReturnMsg
‘io(po,12,y)
y = not y
wait
[rx]
serial2input z$
mi = millis()
m$ = str(mi)
ret = asc(mid(z$,1,1))
if cm == “TAIN1” then ret = m$ & “,” & ret
Msgreturn ret
Return
[tm]
io(pwo,12,(x and 15))
x = not x
wait
```

5.3.5 HTTP-SERVER FÜR DAS CAMFACE (RS232)

Routine zu Abschnitt 0 zum Einsatz des älteren CamFace-Interfaces.

```
memclear
x = -1
y = -1
timer 1000, [tm]
baud = 19200
```

```
Print “RS232 TO WIFI (CAMFACE)”
serial2begin baud, 5, 4 ‘TX/RX
Serial2Print chr(81) ‘DOUT
Serial2Print chr(15) ‘00001111 Ready
Serial2branch [rx]
msgbranch [msgbr]
wait
[msgbr]
cm = upper(msgget(“cmd”))
va = val(msgget(“val”))
MyReturnMsg = “Server CAMFACE”
‘cls
‘Print cm&” “&str(va)
If cm == “LED1” then Io(po,2,0)
If cm == “LED0” then Io(po,2,1)
if cm == “LDR” then
 MyReturnMsg = str(io(ai))
end if
if cm == “SEND” then
 Serial2Print chr(va)
 MyReturnMsg = ““‘str(va
end if
if cm == “HARD” then
 Serial2Print chr(13)
 MyReturnMsg = ””
end if
if cm = “BAUD” then
 Serial2end
 Serial2begin va, 5, 4
 MyReturnMsg = str(baud)
 baud = va
end if
‘+++++CAMFACE++++++++++++++++
if cm == “AIN1” then
 Serial2Print chr(48)
 MyReturnMsg = ””
```

```
end if
if cm == “TAIN1” then
 Serial2Print chr(48)
 MyReturnMsg = ””
end if
if cm == “AIN2” then
 Serial2Print chr(50)
 MyReturnMsg = ””
end if
if cm == “DIN” then
 Serial2Print chr(211)
 MyReturnMsg = ””
end if
if cm == “DOUT” then
 Serial2Print chr(81)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
if cm == “AOUT1” then
 Serial2Print chr(58)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
if cm == “AOUT2” then
 Serial2Print chr(60)
 Serial2Print chr(va)
 MyReturnMsg = str(va)
end if
‘+++++CAMFACE+++++++++++++++++
If MyReturnMsg <> ”” then msgreturn MyReturnMsg
io(po,12,y)
y = not y
wait
[rx]
serial2input z$
mi = millis()
```

```
m$ = str(mi)
ret = asc(mid(z$,1,1))
if cm == “TAIN1” then ret = m$ & “,” & ret
Msgreturn ret
Return
[tm]
io(pwo,12,(x and 15))
x = not x
wait
```

5.3.6 MEHRKANALMESSUNG RC MIT ADS1115

Listing zu Abschnitt 4.6.5: (/btrcadsa0a1.bas)

```
‘RC Entladekurve 2 kanal gpio15 an ads1115
‘Tastervariante
memclear
x = 0
y = 88
vpb = 0.1875
html |A0|
meter y,0,3300
‘textbox y
idy = htmlid()
html |A1|
meter y1,0,3300
‘textbox y1
idy1 = htmlid()
button “R”,[rot]
meter r, -1,0
html | (R)ot schaltet GPIO 15 um (0V-3,3V)<br>|
rot=-1
html |<html><body bgcolor=“whitesmoke”><br>|
html |<canvas id=“myCanvas” width=800 height=320</canvas>|
html |<script src=“http://hjberndt.de/soft/bt93.js” type=“text/javascript”> |
html |</script><script type=“text/javascript”>|
html |var k1 = document.getElementById(“| & htmlvar(idy ) & |”);|
html |var k2 = document.getElementById(“| & htmlvar(idy1) & |”);|
html |const NMAX=100;var filled=false;var tmax=10;|
html |var TabY = new Array(NMAX); |
html |var TabX = new Array(NMAX); |
html |var TabY1 = new Array(NMAX);|
html |function Messen(){var i,x,y;|
html |for(i=1;i<NMAX;i++){|
html |TabY[i-1] = TabY[i]; TabY1[i-1]=TabY1[i];|
html |TabX[i]=i/NMAX*tmax;}|
```

```
html |TabX[NMAX-1]=tmax;TabY[NMAX-1]=k1.value;TabY1[NMAX-1]=k2.value;}|
html |function Anzeigen(){var i; Grafik(AN);Messen();yachse="";xachse="";|
html |titel="ESPBASIC: Echtzeit RC an ADS1115 A0/A1 - Berndt 2019";|
html |farbe ="White" ;cls();farbe="whitesmoke";hintergrund(); |
html |farbe="darkgrey" ; Diagramm(0,tmax,0,0,4000,500); farbe="gray"; |
html |ctx.lineWidth=3;|
html |for(i=0;i<NMAX-1;i++){|
html |farbe=BLAU;DiaLinie(TabX[i],TabY[i],TabX[i+1],TabY[i+1]);|
html |farbe=ROT; DiaLinie(TabX[i],TabY1[i],TabX[i+1],TabY1[i+1]);}}|
html |window.setInterval("Anzeigen()", 200);</script><br></body></html>|
timer 200,[messen]
wait
[messen]
MSB = "C1"'KANAL A0gnd
gosub [config]
gosub [convert]
y = u'Messwert abrufen
bits0 = bits
MSB = "D1"'KANAL A1gnd
gosub [config]
gosub [convert]
y1 = u'Messwert abrufen
bits1 = bits
wait
[config]
i2c.begin(hextoint("48"))'adr
i2c.write(1) 'CONFIG
i2c.write(hextoint(MSB)) 'MSB
i2c.write(hextoint("83"))'LSB
i2c.end()
delay 2
return
[convert]
i2c.begin(hextoint("48"))'adr
i2c.write(0) 'CONVERT
i2c.end()
```

```
i2c.requestfrom(hextoint(“48”),2)
msb = i2c.read()
lsb = i2c.read()
‘Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
return
[rot]
io(po,15,rot)
rot = not rot
wait
```

5.3.7 Differenzmessung A0-A1 durch Mehrkanalmessung gegen Masse

Listing zu Abschnitt 4.6.6 (/btrcdifa0a1man.bas)

```
‘RC Entladekurve 2 Kanal mit Differenzbildung ads1115
‘Tastervariante
memclear
x = 0
y = 88
vpb = 0.1875
html |A0|
meter y,0,3300
‘textbox y
idy = htmlid()
html |A0-A1|
meter y1,-3300,3300
textbox y1
idy1 = htmlid()
button “R”,[rot]
meter r, -1,0
html | (R)ot schaltet GPIO 15 um (0V-3,3V)<br>|
rot=-1
html |<html><body bgcolor=“whitesmoke”><br>|
html |<canvas id=“myCanvas” width=800 height=320</canvas>|
html |<script src=“http://hjberndt.de/soft/bt93.js” type=“text/javascript”> |
html |</script><script type=“text/javascript”>|
html |var k1 = document.getElementById(“| & htmlvar(idy ) & |”);|
html |var k2 = document.getElementById(“| & htmlvar(idy1) & |”);|
html |const NMAX=100;var filled=false;var tmax=10;|
html |var TabY = new Array(NMAX); |
html |var TabX = new Array(NMAX); |
html |var TabY1 = new Array(NMAX);|
html |function Messen(){var i,x,y;|
html |for(i=1;i<NMAX;i++){|
html |TabY[i-1] = TabY[i]; TabY1[i-1]=TabY1[i];|
html |TabX[i]=i/NMAX*tmax;}|
```

```
html |TabX[NMAX-1]=tmax;TabY[NMAX-1]=k1.value;TabY1[NMAX-1]=k2.value;}|
html |function Anzeigen(){var i; Grafik(AN);Messen();yachse="";xachse="";|
html |titel="ESPBASIC: Echtzeit RC an ADS1115 A0/A0-A1 - Berndt 2019";|
html |farbe ="White" ;cls();farbe="whitesmoke";hintergrund(); |
html |farbe="darkgrey" ; Diagramm(0,tmax,0,-4000,4000,1000); farbe="gray"; |
html |ctx.lineWidth=3;|
html |for(i=0;i<NMAX-1;i++){|
html |farbe=BLAU;DiaLinie(TabX[i],TabY[i],TabX[i+1],TabY[i+1]);|
html |farbe=ROT; DiaLinie(TabX[i],TabY1[i],TabX[i+1],TabY1[i+1]);}}|
html |window.setInterval("Anzeigen()", 200);</script><br></body></html>|
timer 200,[messen]
wait
[messen]
MSB = "C1"'KANAL A0gnd
gosub [config]
gosub [convert]
y = u'Messwert abrufen
bits0 = bits
MSB = "D1"'KANAL A1gnd
gosub [config]
gosub [convert]
y1 = y - u 'Messwert abrufen
bits1 = bits
wait
[config]
i2c.begin(hextoint("48"))'adr
i2c.write(1) 'CONFIG
i2c.write(hextoint(MSB)) 'MSB
i2c.write(hextoint("83"))'LSB
i2c.end()
delay 2
return
[convert]
i2c.begin(hextoint("48"))'adr
i2c.write(0) 'CONVERT
```

```
i2c.end()
i2c.requestfrom(hextoint("48"),2)
msb = i2c.read()
lsb = i2c.read()
'Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
return
[rot]
io(po,15,rot)
rot = not rot
wait
```

5.3.8 Differenzmessung A0-A1 mit Taster Timer und Button

Listing zur Differenzmessung aus Abschnitt 4.6.6

```
'RC entladekurve 2 kanal gpio15 an ads1115
'Button Auto Taster
memclear
interrupt 4,[rot]'GPIO 4
auto = -1
vpb = 0.0001875
button "M",[rot]
meter rot, -1,0
button "A",[auto]
html | A0:|
meter y,0,4
'textbox y
idy = htmlid()
html | A0-A1:|
meter y1,-4,4
'textbox y1
idy1 = htmlid()
'textbox auto
html |<br>(M)anuell, (A)uto oder der Taster schaltet GPIO 15 um (0/3,3)V|
rot=-1
html |<html><body bgcolor="whitesmoke"><br>|
html |<canvas id="myCanvas" width=800 height=400</canvas>|
html |<script src= http://hjberndt.de/soft/bt93.js type="text/javascript"> |
html |</script><script type="text/javascript">|
html |var k1 = document.getElementById("| & htmlvar(idy ) & |");|
html |var k2 = document.getElementById("| & htmlvar(idy1) & |");|
html |const NMAX=100;var tmax=10;|
html |var TabY = new Array(NMAX); |
html |var TabX = new Array(NMAX); |
html |var TabY1 = new Array(NMAX);|
html |function Messen(){var i,x,y;|
```

```
html |for(i=1;i<NMAX;i++){|
html |TabY[i-1] = TabY[i]; TabY1[i-1]=TabY1[i];|
html |TabX[i]=i/NMAX*tmax;}|
html |TabX[NMAX-1]=tmax;TabY[NMAX-1]=k1.value;TabY1[NMAX-1]=k2.value;}|
html |function Anzeigen(){var i; Grafik(AN);Messen();yachse="";xachse="";|
html |titel="Messen und Steuern mit ESPBASIC: Echtzeit RC an ADS1115 A1/A0-A1 - Berndt";|
html |farbe ="White" ;cls();farbe="whitesmoke";hintergrund(); |
html |farbe="black" ; Diagramm(0,tmax,0,-4,4,1); farbe="gray"; |
html |ctx.lineWidth=3;|
html |for(i=0;i<NMAX-1;i++){|
html |farbe=BLAU;DiaLinie(TabX[i],TabY[i],TabX[i+1],TabY[i+1]);|
html |farbe=ROT; DiaLinie(TabX[i],TabY1[i],TabX[i+1],TabY1[i+1]);}}|
html |window.setInterval("Anzeigen()", 200);</script><br></body></html>|
timer 200,[messen]
wait
[messen]
MSB = "C1"'KANAL A0gnd
gosub [config]
gosub [convert]
u1 = u'Messwert abrufen
bits0 = bits
MSB = "D1"'KANAL A1gnd
gosub [config]
gosub [convert]
st = 3.3 * abs(io(laststat,(15)))
y1 = st- u
'Messwert abrufen
bits1 = bits
y = u
wait
[config]
i2c.begin(hextoint("48"))'adr
i2c.write(1) 'CONFIG
i2c.write(hextoint(MSB)) 'MSB
i2c.write(hextoint("83"))'LSB
```

```
i2c.end()
delay 2
return
[convert]
i2c.begin(hextoint("48"))'adr
i2c.write(0) 'CONVERT
i2c.end()
i2c.requestfrom(hextoint("48"),2)
msb = i2c.read()
lsb = i2c.read()
'Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
return
[rot]
io(po,15,rot)
rot = not rot
wait
[steuern]
io(po,15,rot)
rot = not rot
return
[auto]
if auto then
 timercb 5000,[steuern]
 gosub [steuern]
 else timercb 0,[steuern]
endif
auto = not auto
wait
```

5.3.9 ASD MIT STEUERBARER SPANNUNGQUELLE MIT UDP

Listing zum Abschnitt 4.6.11

```
print “Regelbare Spannungsquelle <br>”
memclear
rm = 1000
n = 30
html “N: “
textbox n
aout = 511
html “<br>0.0”
slider aout, 0, 1023
vpb = 0.0001875
html |3.3 V<br>PWM = |
textbox aout
html |<hr>|
html “A0 gemessen als Mittelwert [V] “
textbox u0
html |<br>|
html “A0 berechnet aus PWM [V]”
textbox ux
html |<br>|
html “Relativer Fehler [%]”
textbox frel
udpbegin 8080
udpbranch [udp]
timer 5000,[messen]
[messen]
s = 0
io(pwo,15,aout)
for i = 1 to n
 MSB = “C1“‘KANAL A0gnd
 gosub [config]
 gosub [convert]
 s = s + u ‘Messwert abrufen
```

```
next i
 u0 = s / n
 ux = 3.3 * aout / 1023
 frel = (ux - u0) / ux * 100
 temp = replace(str(frel),".",",")
for cl = 20 to 34
 u$ = "192.168.178."&str(cl)
 udpwrite u$,8080,temp
next cl
wait
MSB = "D1"'KANAL A1gnd
gosub [config]
gosub [convert]
um = u 'Messwert abrufen
ux = u0 - um
rx = rm * ux / um
wait
[config]
i2c.begin(hextoint("48"))'adr
i2c.write(1) 'CONFIG
i2c.write(hextoint(MSB)) 'MSB
i2c.write(hextoint("83"))'LSB
i2c.end()
delay 2
return
[convert]
i2c.begin(hextoint("48"))'adr
i2c.write(0) 'CONVERT
i2c.end()
i2c.requestfrom(hextoint("48"),2)
msb = i2c.read()
lsb = i2c.read()
'Spannung berechnen mit Vorzeichen
bits = msb *256 + lsb
if bits>32767 then bits=bits-65536
u = bits * vpb
```

```
return
[udp]
a = udpread()
udpreply frel
return
```

5.3.10 STEUERBARE SPANNUNGSQUELLE

Listing zu 4.6

```
print “ADS1115 A0/GND PWM RC<br>”
i2c.setup(0,2)
vpb = 0.0001875’ Gain 0.67
slider aout,0,1023
textbox uout
wprint ” Steuerung 0..3,3 Volt<hr>”
textbox bits
wprint ” Bits (“&vpb&” Volt/Bit)<br>”
textbox u
wprint ” U/V<br>-5……0……+5<br>”
meter u,-5,5
timer 500 ,[messen]
timercb 200,[cb]
wait
[cb]
io(pwo,5,aout)
uout = int(aout / 1023 * 3.3 * 100)/100
return
[messen]
‘Konfigurieren
i2c.begin(hextoint(“48”))‘adr
i2c.write(1) ‘CONFIG
i2c.write(hextoint(“c1”))‘MSB
i2c.write(hextoint(“83”))‘LSB
i2c.end()
‘Messwert abrufen
i2c.begin(hextoint(“48”))‘adr
i2c.write(0) ‘CONVERT
```

```
i2c.end()
i2c.requestfrom(hextoint("48"),2)
msb = i2c.read()
lsb = i2c.read()
'Spannung berechnen mit Vorzeichen
bits = msb * 256 + lsb
if bits > 32767 then bits = bits - 65536
uu = bits * vpb
u = int(uu*100)/100
wait
```

5.3.11 ESP-ROUTINEN ZUR GRENZFREQUENZ

Routinen zur Sukzessiven Approximation in 4.8.4 (Cover)

```
Memclear
MOSI = 13
CL = 14
RS = 15
FO = 16
Textbox f
Button " Manuelle Frequenz",[goset]
textbox a
wprint " u<br>"
Textbox c
wprint " u/u0<br>"
Textbox a0
wprint " u0"
spi.setup(1000000,0,0)
p = 0
f = 500
a0 = -1
N = 20
gosub [init]
t = "Sukzessive Approximation"
stat = t
Wprint "<hr>"
Textbox R
```

```
Wprint ” Ohm (Vorgabe)<br>”
Textbox C
Wprint ” uF (Messung)<hr>”
Textbox mn
Wprint ” fmin<br>”
Textbox mx
Wprint ” fmax<br>”
Textbox cnt
wprint ” Messungen<br>”
Textbox stat
Button “Reset”,[reset]
Print
button “approximieren”,[apx]
[reset]
cnt = 0
mn = 100
mx = 10000000
stat = t
f = 100
a0 = -1
[goset]
gosub [set]
Wait
[apx]
C = “mal sehen…”
stat = “suche Grenzfrequenz…”
st = (mx - mn) / N
ff = mn
f = mn
[wdh]
 gosub [set]
 gosub [messen]
 if u <= 0.707 then
  mx= f ‘neue Grenzen
  mn = ff
  goto [exit]
```

```
 endif
 ff = f
 f = f + st
goto [wdh]
[exit]
if int(mn) == int(mx) then
 stat = int(mx)&" Hz gefunden"
 C = 1000000 / (2 * pi * mx * R)
 wait
endif
if cnt < 300 then goto [apx]
stat = "Kein Ergebnis"
wait
[init]
C = 0.047
R = 1000
pi = acos(-1)
a0 = -1
return
[messenalt]
If f<= 0 then f = 1
Xc = 1 / (2 * pi * f * C * 0.001 * 0.001)
phi = atan(R / Xc)
u = cos(phi)
cnt = cnt + 1
return
[messen]
max = 0
min = 5
for j = 1 to 50
 aa = io(ai) / 310
 if aa > max then max = aa
 If aa < min then min = aa
Next j
a = (max - min)
if a0 < 0 then a0 = a
```

```
c = (a / a0)
u = c
cnt = cnt + 1
return
[set]
dp = f * 4294967296 / 125000000
ph = p << 3
for byte = 1 to 4
 b = dp and 255
 spi.byte(b)
 dp = dp >> 8
next byte
spi.byte(ph)
pin = FO
gosub [pulse]
return
[pulse]
io(po,pin,1)
io(po,pin,0)
return
```

5.3.12 Frequenzgenerator AD9850 über I2C

Arduino Routinen in C zu Abschnitt 4.9.2.

```
#include <Wire.h>
/* I2C Lösung mit UNO für ESPBASIC
  * ARDUINO SDA A4 SCL A5
*/
//http://www.vwlowen.co.uk/arduino/AD9850-waveform-generator/AD9850-waveform-generator.htm
#define AD9850_CLOCK 125000000
// AD9850 Module pins Arduino uno
#define SW    8
#define W_CLK 9
#define FQ_UD 10
#define DATA  11
#define RESET 12
```

```
#define LED   13
#define pulseHigh(pin) {digitalWrite(pin, HIGH); digitalWrite(pin, LOW); }
void tfr_byte(byte data)
{for (int i = 0; i < 8; i++, data >>= 1)
 {digitalWrite(DATA, data & 0x01);
  pulseHigh(W_CLK);
 }
}
void sendFrequency(double frequency)
{int32_t freq1 = frequency * 4294967295/AD9850_CLOCK;
 for (int b = 0; b < 4; b++, freq1 >>= 8)tfr_byte(freq1 & 0xFF);
 tfr_byte(0x000);
 pulseHigh(FQ_UD);
}
void setup()
{pinMode(FQ_UD, OUTPUT);
 pinMode(W_CLK, OUTPUT);
 pinMode(DATA,  OUTPUT);
 pinMode(RESET, OUTPUT);
 pinMode(SW, INPUT);
 Wire.begin(8);   // join i2c bus with address #8
 Wire.onReceive(receiveEvent);
 Serial.begin(115200);
 Serial.println("AD9850 i2c Slave 0x08 init 1 MHz");
 sendFrequency(1000000);
 sendFrequency(1000000);
}
void loop() {}
void receiveEvent(int howMany)
{double ff;unsigned long f;
 byte t0,t1,t2,t3;
 if(Wire.available())
 {t3 = Wire.read();
  t2 = Wire.read();
  t1 = Wire.read();
  t0 = Wire.read();
```

```
  ff = (t3*16777216L)+(t2*65536L)+(t1*256L)+t0;
   sendFrequency(ff);
  sendFrequency(ff);
  Serial.println(ff);
 }
}
```

5.3.13 RFO-BASIC ESP-ROUTINEN ZUR KENNLINIE.BAS

Erweitertes Listing zu Abschnitt 4.10

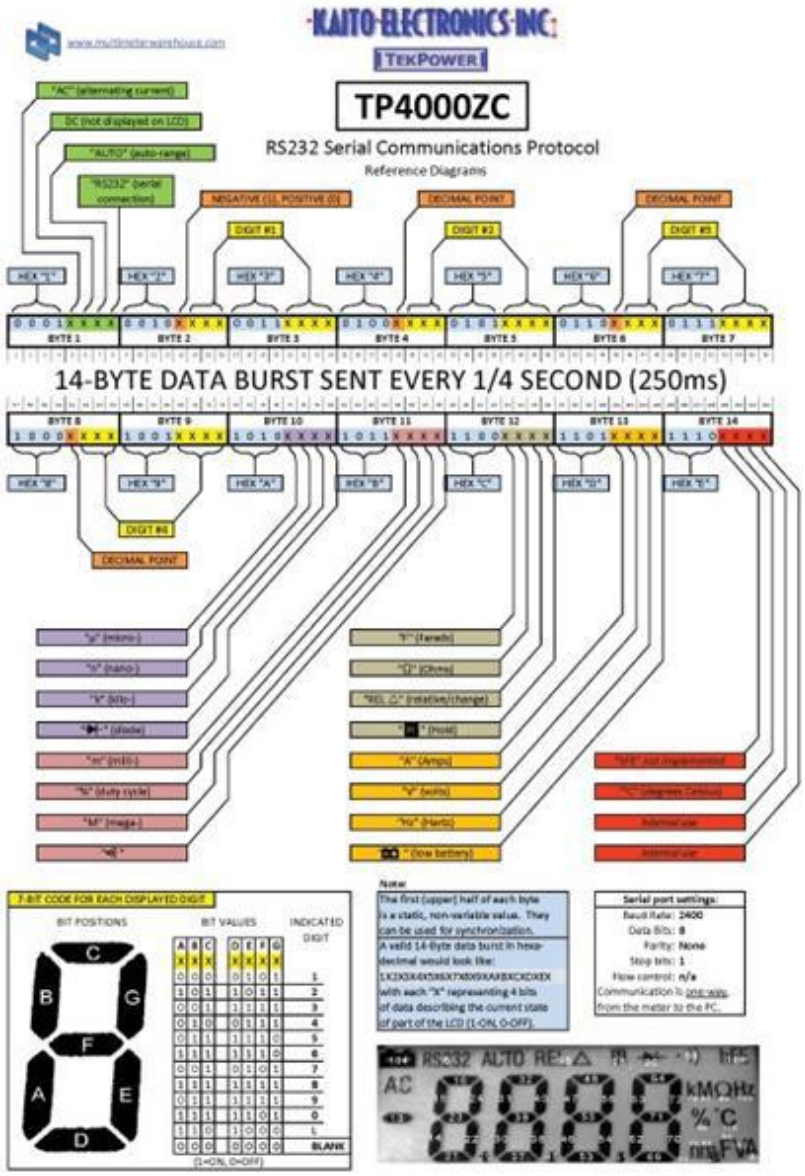
```
FN.DEF url$()
!FN.RTN “http://192.168.178.39/”
FN.RTN “http://192.168.4.1/”
!FN.RTN “http://192.168.43.101/”
FN.END
FN.DEF DOUT(B)
 cmd$=“msg?cmd=dout&val=”+INT$(b)
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF AIN(B)
 cmd$=“msg?cmd=ain”+INT$(b)
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF TAIN(B,t)
 cmd$=“msg?cmd=tain”+INT$(b)
 GRABURL r$, url$()+cmd$
 i=IS_IN(“,”,r$)
 t=VAL(LEFT$(r$,i-1))
 FN.RTN VAL(MID$(r$,i+1))
FN.END
FN.DEF AOUT(B,C)
 cmd$=“msg?cmd=aout”+INT$(b)+”&val=”+STR$(c)
 GRABURL r$, url$()+cmd$
```

```
 !r$="12"
 FN.RTN VAL(r$)
FN.END
FN.DEF DIN()
 cmd$="msg?cmd=din&val=0"
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF Send(b)
 cmd$="msg?cmd=send&val="+INT$(b)
 GRABURL r$, url$()+cmd$
 if r$="ok" then r$=str$(b)
 FN.RTN val(r$)
FN.END
FN.DEF Read()
 cmd$="msg?cmd=read"
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF ReadClab()
 cmd$="msg?cmd=read&val=9"
 GRABURL r$, url$()+cmd$
 FN.RTN val(r$)
FN.END
FN.DEF Cmd(c$)
 GRABURL r$, url$()+c$
 IF r$="ok" THEN r$="-1"
 FN.RTN VAL(r$)
FN.END
FN.DEF flush()% nur c++
 WHILE read()>=0:REPEAT
FN.END
FN.DEF ip$()
```

```
 a$=MID$(url$(),8)
 FN.RTN REPLACE$(a$,"/","")
FN.END
FN.DEF iain(b)
 SOCKET.CLIENT.CONNECT ip$(), 80
 SOCKET.CLIENT.WRITE.LINE "/msg?cmd=ain"+INT$(b)
 maxclock = CLOCK() + 10000
 DO
  SOCKET.CLIENT.READ.READY flag
  IF CLOCK() > maxclock
   PRINT "Read time out"
   END
  ENDIF
 UNTIL flag
 !SOCKET.CLIENT.SERVER.IP s$
 SOCKET.CLIENT.READ.LINE l1$
 SOCKET.CLIENT.READ.LINE l2$
 SOCKET.CLIENT.READ.LINE l3$
 SOCKET.CLIENT.READ.LINE l4$
 !PRINT l1$,l2$,l4$
 FN.RTN VAL(l4$)
 SOCKET.CLIENT.CLOSE
FN.END
FN.DEF Baud(rate)
 cmd$="msg?cmd=baud&val="+int$(rate)
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF Hard()
 cmd$="msg?cmd=hard"
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
FN.DEF Led(an)
 cmd$="msg?cmd=led"+int$(an)
```

```
 GRABURL r$, url$()+cmd$
 FN.RTN an
FN.END
FN.DEF freq(f)
 cmd$="msg?cmd=freq&val="+int$(f)
 GRABURL r$, url$()+cmd$
 FN.RTN f
FN.END
FN.DEF Ldr()
 cmd$="msg?cmd=ldr"
 GRABURL r$, url$()+cmd$
 FN.RTN VAL(r$)
FN.END
! TESTING
GOTO main
send(1)
PRINT read()
main:
?ip$()
!led(0)
!freq(440)
!call dout(1)
!aout(1,128)
!?ain(1)
```

5.3.1 Datenblatt zur Dekodierung des Multimeters



https://tekpower.us/media/catalog/product/cache/1/image/650x/040ec09b1e35df139433887a97daa66f/t/p/tp4000zc_serial_protocol.jpg

---

Literaturverzeichnis

---

[1] Messen mit dem Smartphone , H.-J. Berndt, Eigene Programme auf Android Tablet und Phone, Kindle-eBook

ASIN B00CO5TGEK, Mai 2013

[2] **Messen und Steuern mit dem Smartphone** , H.-J. Berndt, Bluetooth, USB, RS232, Arduino mit Android Tablet/Phone, Kindle-eBook ASIN B00SM1UMQG, Januar 2015

[3] **Messen Steuern Regeln mit dem Smartphone und Tablet** , H.-J. Berndt, TCP/IP, WiFi, Bluetooth, USB, RS232 im Zusammenspiel mit Android, Windows, ESP8266, Digispark, Arduino u.a., Kindle-eBook ASIN B075HLRBYP, September 2017

[4] **Messen, Steuern und Regeln mit Word und Excel,** H.-J. Berndt / B. Kainka, VBA-Makros für die serielle Schnittstelle, 3., aktualisierte Auflage, Franzis-Verlag GmbH, 85586 Poing, 2001, ISBN 3-7723-4094-6
bzw. der Nachdruck dieser Auflage

[5] ESP8266BASIC Referenz via
https://docs.google.com/document/d/1EiYugfu12X2_pmfmu2O19CcLX0ALgLM4r2YxKYyJon8/pub

und

https://docs.google.com/document/d/1EiYugfu12X2_pmfmu2O19CcLX0ALgLM4r2YxKYyJon8/export?format=pdf
abgerufen Ende 2019

[1] https://github.com/esp8266/Arduino/issues/4061