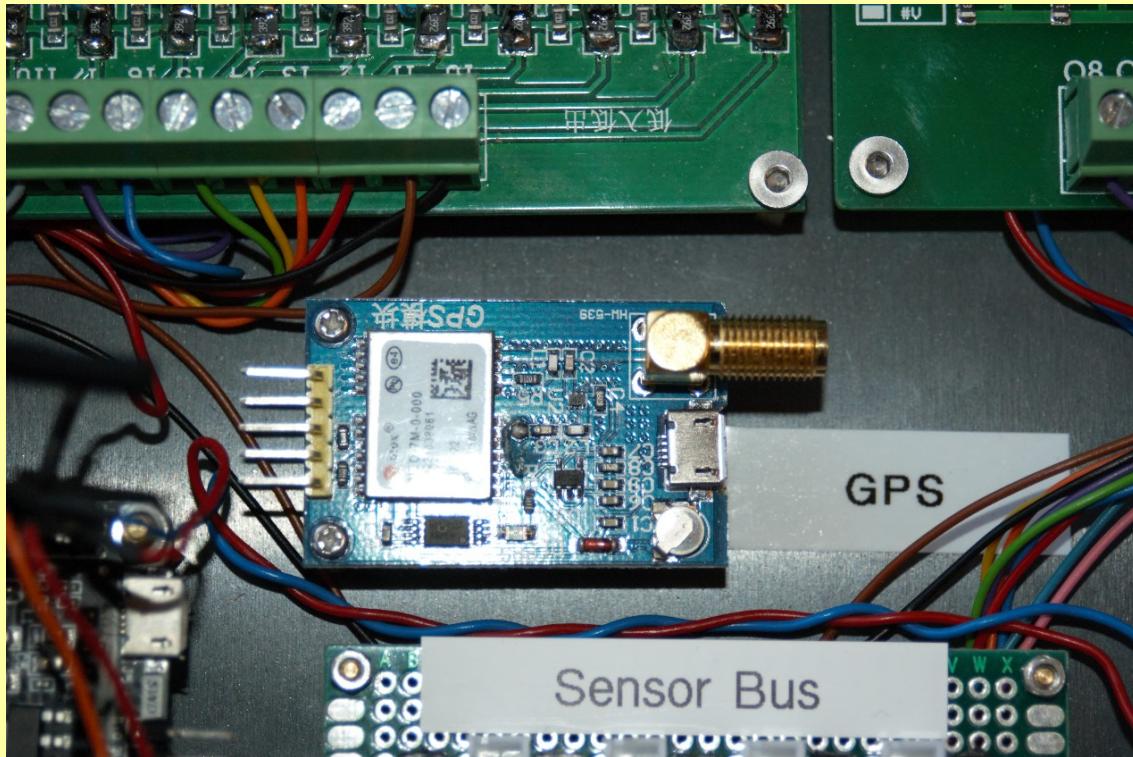


# pahlbasic for arm

01/2025

V1.15b

**Messen, Steuern, Regeln mit  
Basic wie in den 80ern – jetzt auf  
aktueller ARM Hardware**



# Inhaltsverzeichnis

Inhaltsverzeichnis.....	1
Was ist pahlbasic?.....	11
Bedienung.....	13
Basic Anweisungen.....	15
array / aR.....	15
baud.....	15
break.....	16
case.....	16
call, rcall, lcall.....	17
cd.....	18
celse.....	18
clear / cL.....	19
close.....	19
cls.....	19
config / cO.....	20
cont.....	27
convert / conV.....	27
data.....	28
debug / deB.....	28
dec, decr.....	29
deffn.....	29
del.....	29
dir.....	30
do.....	31
draw.....	31
dump / dU, rdump.....	34
else.....	35
end, end.....	35
endif.....	35
erase.....	35
exit.....	36
fkey.....	36
for.....	36

format.....	36
function / fU.....	37
get.....	37
get adx#, get diginp#, get input#, get output#, get relay#.....	37
get dir\$.....	38
get #.....	39
goto / gO.....	40
gosub / goS.....	40
gps.....	41
i2c.....	42
i2c2.....	43
if.....	44
impuls / iM.....	45
inc, incr.....	45
include.....	46
input / inP.....	47
input #.....	47
input bin.....	48
input hex.....	48
inpt.....	48
insert / inS.....	49
jmp.....	49
keypad / keY.....	49
kill.....	50
lcd.....	50
led.....	51
let.....	52
list / ll, rlist.....	53
load / IO, rload.....	54
locate / loC.....	54
max.....	54
md.....	55
mean.....	55
midi.....	56

min.....	56
morse / mO.....	56
new.....	56
next / nE.....	57
on.....	57
play.....	63
plot.....	64
poke / pO.....	64
print, ? / pR.....	65
print #.....	66
proc.....	67
put.....	67
put #.....	67
ram, rom.....	68
rcall.....	68
rd.....	68
read.....	68
receive / reC.....	69
record.....	70
rem, '.....	71
rename / reN.....	71
reset.....	71
reset #.....	72
restore / reS.....	73
resume.....	73
return / reT.....	74
rload.....	74
run, rrun.....	74
save / sA.....	75
seed.....	76
select / seL.....	77
selend.....	78
send.....	78
server.....	79

set.....	80
set #.....	82
sleep.....	83
sort.....	83
sound / sO.....	84
spi.....	85
stamp.....	85
start, !, rstart.....	85
sub.....	86
text.....	86
then.....	86
toggle / tO.....	86
touch.....	87
until / uN.....	87
usb.....	87
user.....	87
wait.....	88
wdog.....	89
while wend.....	89
write.....	89
 Variablen.....	90
numerische Variablen.....	90
Arrays.....	90
Zeichenketten Variablen.....	93
Text Array.....	93
Systemvariablen.....	94
ad1.....	94
ad2.....	94
ad3.....	94
ad4.....	94
attack.....	94
buf.....	94
clock / clo.....	95
color / col.....	95

dac.....	95
dataptr / daT.....	95
date.....	96
day.....	96
decay.....	96
diginp / diG.....	96
dow.....	97
erl.....	97
err.....	97
fnvar, result / fN, rE.....	97
font.....	97
hour.....	97
intrpt.....	97
minute / miN.....	97
milli.....	98
month / moN.....	98
pgm.....	98
pwm1, pwm2, pwm3, pwm4.....	98
relay / reL.....	99
rand.....	99
release / rL.....	99
second / seC.....	99
speed.....	99
stack.....	100
sustain / suS.....	100
tempo / teM.....	100
time.....	100
urgend.....	100
volume / vO.....	100
year.....	100
Zugriff auf einzelne Bits.....	101
Operatoren.....	102
arithmetisch.....	102
logisch.....	105

!	105
and, &&	105
or,	105
xor,	105
Vergleich	106
>	106
<	106
>=	106
<=	106
=	106
<>	106
Funktionen	107
abs()	107
atn()	107
calc()	107
cos()	108
epoch()	108
fn1() ...fn5()	109
fn Name(), fr Name()	109
fn Attrib()	109
fn Choice()	109
fn Dir()	109
fn Eof()	110
fn Fileptr()	110
fn Fourier()	110
fn Fmod()	111
fn Midi()	111
fn Open()	112
fn Peek()	113
fn Pointer()	113
fn Rgb()	113
fn Rgb2()	114
fn Red()	114
fn Green()	114

fn Bue()	114
fn Size()	115
fn Testbit()	115
high()	115
int()	115
low()	115
log()	115
norm()	116
not()	116
num()	116
peek()	116
sensor()	116
sign()	118
sin()	118
sqr()	118
sys()	119
tan()	119
val()	119
User definierte Funktionen1	120
User definierte Funktionen2	121
Stringfunktionen	123
( )	123
asc()	123
bin\$()	123
chr\$()	123
cls\$	124
color\$()	124
count\$()	124
dir\$	125
dow\$	125
epoch\$()	125
f\$, fs	125
fix\$()	126
fkey\$()	126

gps\$.....	126
hex\$().....	127
in.....	127
inkey.....	127
inkey\$.....	128
&, +, _, \.....	128
lcase.....	129
len().....	129
loc\$().....	129
mid.....	129
mpeek().....	130
path\$.....	130
prog\$.....	130
result\$.....	130
sensor\$.....	130
stamp\$.....	130
str\$().....	131
string().....	132
ucase.....	132
Vergleiche.....	133
=, <>, >, <, in.....	133
Einzelnes oder mehrere Zeichen (Teilketten).....	134
einzelnes Zeichen:.....	134
Position Null.....	134
Teilzeichenkette.....	135
Textarray.....	135
Konstanten.....	136
true.....	136
false.....	136
pi.....	136
e.....	136
Zahlen.....	137
Hexadezimalzahlen.....	137
Binärzahlen.....	137

Fehlermeldungen.....	138
Unterschiede zu anderen Tiny Basic-Dialekten.....	140
Beispiele für Syntax-Abweichungen:.....	141
Neu in pahlbasic.....	142
Anhang 1.....	143
Toucheingabe mit der keypad-Anweisung:.....	143
Der Info-Screen im Direktmodus.....	144
Die Funktion sys().....	145
Der Einzelschritt-Modus:.....	149
Die Eingabezeile.....	150
Funktionstasten.....	151
Farben & Fonts.....	151
Vom System vordefinierte Farben.....	151
VT100-Farben:.....	152
Fonts.....	152
Autostart.....	152
Bits und Bytes.....	153
Programmausführung im ROM.....	154
Hardware/Firmware.....	155
Der STM32F407 Controller.....	155
Ein Wort zum Anschluss externer Komponenten.....	155
Wie kommt das Basic in den Controller-Chip?.....	156
Die Boards.....	157
407-Mini-Board.....	157
407-VE-Board.....	158
Black Pill.....	159
Sensoren.....	160
Anschluss- / Schaltpläne.....	161
1.) Das 407-Mini Board:.....	161
2.) Das 407ZG-Board.....	163
3.) BlackPill.....	164
Erste Schritte mit dem Basic-Controller.....	166
4.) Das DIY-More-Board.....	167
record-Pin-Anschluss.....	168

Port-Ausgänge.....	169
Der Sensor-Bus.....	170
Audio-Anschluss:.....	170
DS18x20 One-Wire-Temperatur-Sensoren-Bus.....	171
USB-Keyboard.....	172
SD-Card.....	173
MIDI.....	174
Modbus.....	175
Vom Basic nutzbare Hardware-Interrupts.....	179
Datei senden.....	180
Haftungsausschluss.....	181

## Was ist pahlbasic?

pahlbasic ist eine einfache Programmiersprache für einfache Steuerungsanwendungen. Es besteht aus einer Teilmenge von Basic mit hardwarespezifischen Erweiterungen. Es orientiert sich am MCS Tiny Basic (im Intel 8052AH) sowie am Dartmouth-Basic von 1964. Also Vintage Basic. Inzwischen sind moderne Sprachelemente hinzugekommen. Der in pahlbasic geschriebene Code wird interpretiert und ist daher nicht für Anwendungen geeignet, die hohe Geschwindigkeit erfordern. (typ. Ausführungs-Zeit: 5-40µs/Anweisung).

Die Zeile: `10 let time = 0 : do : inc A : until time=10` wird ca. 35000 mal je Sekunde ausgeführt. Zum Vergleich: eine einfache Schleife nur aus `for` und `next` dauerte beim Commodore VC20 (1MHz) ca. 1ms je Durchlauf. pahlbasic ist ca. 200mal schneller. Typische Anwendungen sind einfache Steuerungen oder Regelungen zum Beispiel Alarmsysteme, Anzeigeeinheiten für Messwerte, Schaltuhren, Thermostate oder ähnlich. Mit pahlbasic ist es möglich, mit wenigen Programmierschritten eine einfache Anwendung zu erzeugen, ohne sich in umfangreiche Programmiersprachen einarbeiten zu müssen.

pahlbasic wurde von einem Elektroniker mit Hang zur Nostalgie für kleine praktische Hobby-Spaß-Projekte geschrieben. Man kann vieles besser mit einem Arduino erledigen, aber kann man damit interaktiv probieren, experimentieren und alte Zeiten aufleben lassen?

Ursprünglich für die ATMEGAS entwickelt nun portiert auf den ARM STM32F4-Controller mit 100 oder 168 MHz mit integriertem DAC (F405/7) und RTC, die DCF-Antenne wird nicht mehr unterstützt. Das simple Basic wurde beibehalten.

Die [Hardware](#), die von pahlbasic unterstützt wird, ist ein Experimentierboard mit [16 digitalen](#)- [3 PWM](#)- zwei Analog-Ausgängen die auch der [Soundausgabe](#) dienen. Eingänge sind: [12 Digitaleingänge](#) sowie [4 analoge Spannungsmesseingänge](#). Eine typische Mikro-SPS-Konfiguration. pahlbasic unterstützt weiter ein [TFT-LCD](#)-Display, [mit Touch](#) für Eingaben, einen Anschluss für [GPS](#)-Antenne sowie mehrere [1-Wire](#)-Temperatursensoren. Via [I<sup>2</sup>C](#) sind weitere [Sensoren](#) anschließbar. Siehe [Hardware](#).

Seit der Version 1.04 wird auch ein zusätzlicher [Flashspeicher](#) mit SPI-Anschluss und 2Mbyte - [8MByte](#) mit einem sehr einfachen Datei-System unterstützt, die Version 1.14 bringt [On-Chip-USB](#)-Anschluss, [SD-Card](#), [Bitmaps](#) auf dem LCD, Wiedergabe von [Wave-Dateien](#), In/Out-Erweiterung mit [Modbus](#)-Modulen sowie echte [Basic-Interrupts](#).

Der Programmcode von pahlbasic ist sehr kompakt. Der Programmspeicher der Version für ARM fasst ca. 64KByte im RAM / 256KByte im ROM. Das reicht für mittlere Projekte (ca. 1000-4000 Programmzeilen). Eine Zugangskontrolleinheit mit Tasten-Code ist mit ca. 20 Programmzeilen ruckzuck erstellt.

```
t Codierung Sprache Einstellungen Werkzeuge Makros Ausführen Plugins Fenster ?
RC.C CRC8.C s1531.cpp hal_ll_i2c.c font.c __Lib_System_4XX.c Clock_Test.PBAS
1 proc Anzahl_Satelliten
2   ' GPS-Modul angeschlossen? GPS-Parsing eingeschaltet?
3   print "Anzahl Satelliten: "; fn_Satellites()
4 end.
5
6 function Komma(A, D) : 'A=Anzahl Kommas, D=ZeichenOffset
7   let B = length(@A$) : let E = 1
8   while E<B
9     if A$(E) = "," then inc C
10    if C=A then exitdo
11    inc E
12  wend
13  let result = E+D
14 end
15
16 function Satellites()
17   let A$ = gpa$("GGA")
18   let result = ((A$(fn_Komma(7, 1))-48)*10+(A$(fn_Komma(7, 2)))-48
19 end
```

Abb. 2 So sieht ein pahlbasic-Programm-Text in Notepad++ aus. ([siehe auch: dark](#))

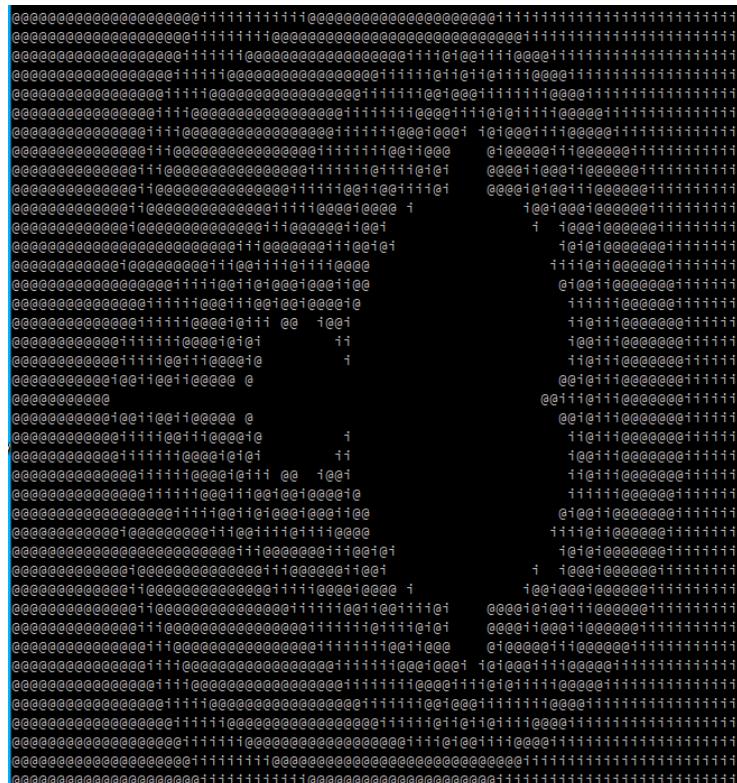
Die [Programmierung](#) wird mit einem Terminal oder PC mit Terminal-Emulator-Programm durchgeführt.

Programme kann man natürlich auch mit jedem Text- oder Programmeditor schreiben und mit TeraTerm (Datei \ Datei senden) vom PC in den Basic-Computer hochladen. ([sogar ohne Zeilen-Nummern!](#))

Es gibt zwei Betriebsmodi: Nach dem Einschalten befindet sich der Basic-Computer im **Direktmodus** = Kommando-Modus. Hier können Anweisungen wie `print time` (wird sofort ausgeführt) oder auch Programmzeilen (beginnen mit einer Zeilennummer) wie: `10 let B = sin(A)` (wird im RAM gespeichert) eingegeben werden. Mit der Anweisung "run" wird das Programm im RAM gestartet und der Basic-Computer wechselt in den **Programm-Modus**. Es gibt Anweisungen, die funktionieren nur im Kommando-Modus und andere nur im Programm-Modus. Viele Anweisungen arbeiten in beiden Modi. Zur Eingabe verfügt pahlbasic über einen rudimentären "Zeileneditor" wie MCS51 Basic. - zusätzlich: **Cursor links** und **rechts**, **Pos1**, **Ende** sowie Überschreiben – auf Terminalprogrammen mit VT100 Unterstützung (empfohlen: TeraTerm) auch Einfügemodus (mit **Strg + e** einschalten) ausserdem **Tabulatoren** direkt hinter der Zeilen-Nummer für Einrückungen.

Das fertige Programm kann dauerhaft im internen oder externen Flash-ROM sowie auf der SD-Card gespeichert werden. Der Basic-Computer wird durch eine Autostartfunktion zum Controller. Das Programm wird dann nach Einschalten des Basic-Computers sofort gestartet.

Let's start, ... goto Seite 13



von **Apfel.PBAS** im Terminalfenster erzeugt

## Bedienung

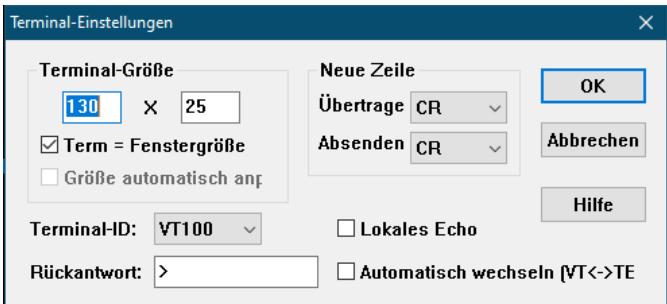


Abb. 2 Terminal-Einstellungen

Der Basic-Computer wird entweder mit einem USB-Kabel oder aber über ein Kabel mit integriertem USB-Seriell-Wandler mit dem PC verbunden. Es wird ein Terminal-Programm wie [TeraTerm](#) benötigt. Einstellungen siehe links.

Den Anschluss bitte dem [Anhang](#) entnehmen.

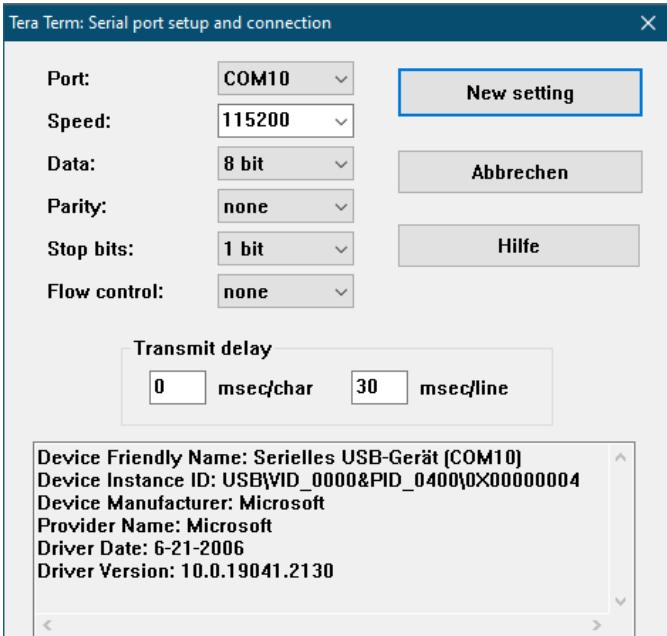


Abb. 3 Serial Port Einstellungen

Den COM-Port notfalls bitte im Gerätetypen manager nachschauen.

Hintergrundfarbe sollte **Schwarz** sein, die Schrift sollte ein Font mit festem Zeichenabstand (z.B. Source Code Pro) sein.

Nach dem Einschalten der Betriebsspannung erscheint die Startmeldung ähnlich der Abbildung rechts

```
USB Keyboard detected
uart1 started
Backup RAM OK
Clock started
found ser. Flash 2MByte
Scanning Sensor-Bus:
Hex Adress: 0040 HTU31
Hex Adress: 0050 24CXX
Hex Adress: 0076 BME280/BMP280 *
found 3 I2C-Devices, * with extra start procedure
started Sensor Bus
started 1 I2C-Devices
SD_Card detected
MCU-ID: 0413 Flash size: 1024 KB
hardware configuration ready, Hardware-Errors: 0
pahlbasic V1.05A31e by Thomas Pahl
75000 bytes free
letzte Nutzung: Sa, 02.12.2023 13:55:09 Uhr
>-
```

Abb. 4 Einschaltmeldung

Eine Programmzeile beginnt immer mit einer Zahl, der Zeilennummer, gefolgt von den Anweisungen und ihren Parametern. Die Programmzeile wird im RAM gespeichert. Weiter

>10letA=55  
passiert zunächst nichts. Ob eine Programmzeile "verstanden" wurde, sieht man sofort nach der Eingabe (wurde der grobe Syntax-Check bestanden, wird die Zeile farblich geändert). Zu jeder Anweisung gehört eine Syntax, die genau eingehalten werden muss.

Sie ist der folgenden Anleitung zu entnehmen. Fehlerhafte Eingaben führen zu Fehlermeldungen. Leerzeichen zwischen den einzelnen Komponenten einer Programmzeile können (sollen) weggelassen werden. Es können mehrere Anweisungen in eine Programmzeile gepackt werden, durch Doppelpunkt getrennt.

Ohne Zeilennummer werden die eingetippte Anweisungen sofort ausgeführt. Dies ist für das Debugging wichtig, da Variablen angezeigt werden können (`print A`) oder verändert (`let B = 9`). Die Programmausführung kann jederzeit durch Eingabe von `Strg + c` auf dem Terminal unterbrochen und durch Eingabe von '`cont`' fortgesetzt werden. Mit `Strg + e` wird der `Einfügemodus` eingeschaltet (gilt nur bis zum Zeilenende). Mit `Strg + p` holt man die letzte Eingabe wieder in den Eingabepuffer, ebenso nach „`list zeile`“ z.B. `list 100`. `Strg + l` löscht die Eingabezeile.

Die folgende Beschreibung der Basic-Befehle ist keine Einführung in die Programmiersprache Basic. Es gibt komplexe und einfache Elemente. Am Besten mit den einfachen anfangen. Es lassen sich funktionierende Programme mit ganz einfachen Mitteln schreiben.

```
10 print "Hello World"  
20 goto 10                                das geht immer noch  
  
10 do  
20   print "Hello World"  
30 until                                    das geht aber auch
```

Auch wenn es um Elektronik geht, wird ein gewisses Grundwissen vorausgesetzt. Im Anhang gibt es zusätzliche Beschreibungen, die noch vervollständigt werden. Work in Progress...

1040 if C=A then exitdo  
1050 inc E  
1060 wend  
1070 let result = E+D  
1090 end  
1100 '  
2000 function Satelites()  
2010 let A\$ = gps\$("GGA")  
2020 let result = ((A\$(fn Komma(7, 1)))-48)\*10+(A\$(fn Komma(7, 2)))-48  
2090 end  
ready  
>

Ein Programm auf TeraTerm

## Basic Anweisungen

Alle [Basic-Anweisungen](#) werden *klein* geschrieben – Ausnahme: für einige Anweisungen gibt es jetzt Abkürzungen, die Grossbuchstaben enthalten (wie bei Commodore: der 2., 3. oder 4. Buchstabe wird grossgeschrieben und verkürzt die Anweisung auf 2, 3 oder 4 Zeichen). Leerzeichen zwischen den Elementen sind nicht nötig – sollten besser weggelassen werden – der Parser kommt damit klar. Der grobe Syntax-Check direkt nach der Eingabe geht sogar davon aus (gilt für den Zeileneditor im Terminal-Fenster). Gelistet wird das Programm immer schön lesbar. Nur [Variablen](#)-Bezeichner werden grossgeschrieben. Siehe [Variablen](#).

Parameter in [ ] können weggelassen werden, alternative Parameter sind durch | getrennt.. Eine Basic-Programmzeile sieht so aus:

**Zeilennummer Befehl 1** [Parameter] [ : **Befehl 2** Parameter] [ : **Befehl 3** Parameter : .... ]  
Max.123 Zeichen.

**ex:**

100 for N=1 to 10 : print N : next

so werden Programmteile dargestellt

Wo immer eine Zahl als Argument erwartet wird, kann auch ein komplexer Rechenausdruck stehen (außer: [list](#), [data](#), [plot](#)). Rechenausdrücke in Parameterangaben besser in Klammern einschließen – entsprechende Hinweise bei der jeweiligen Komponente. Die Farbgebung der folgenden Beschreibung ist dem Syntax-Highlighting von pahlbasic nachempfunden:

**ex. :**

10 gosub 300+(10\*N) ist möglich!

Zeilennummern sind von 1 bis 65535 zulässig.

array / aR

Array mit Werten füllen:

array Startelement, Wert1, Wert2, Wert3, ...

array I(5), 22.5, 330, 2\*pi, 75.0, F, adl

Füllt das Array I() ab Element 5 fortlaufend mit den anschließenden Werten.  
Max. 32 Werte. Siehe auch: [Arrays](#)

baud

Übertragungsgeschwindigkeit temporär einstellen

baud 19200

so werden Kommandos dargestellt

ändert die [Übertragungsgeschwindigkeit](#) der seriellen Konsolen-Schnittstelle. Standard nach Reset ist 115200 Baud (Bits pro Sekunde). [Übertragungsgeschwindigkeit](#) darf folgende Werte annehmen: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 31250. Die letzten Nullen dürfen auch weggelassen werden.

[Übertragungsgeschwindigkeit](#) darf auch ein berechenbarer Ausdruck sein. Wird während einer Terminalsitzung die Übertragungsgeschwindigkeit einseitig geändert, reißt die Verbindung zum Terminal ab. Das Terminal muss ebenfalls auf den neuen Wert eingestellt werden.

Die Anweisung `baud` ändert die Baud-Rate nur temporär (bis zum nächsten Reset), für eine dauerhafte Einstellung bitte die Anweisung [config](#) benutzen.

Für die Konsolenverbindung über den USB-Anschluss (auf dem µC-Board) ist die Einstellung ohne Wirkung. `baud` funktioniert auch im Programm.

## break

`break` dient zum vorzeitigen Verlassen einer Programmschleife. (Bitte Schleifen nicht mit `goto` verlassen.)

### 1.) `break` ohne Parameter

Das Programm wird hinter der *zugehörigen* `wend`-, `until`- oder `next`-Anweisung fortgesetzt:

```
ex.:
10 let A = 0
20 do
30   if A>5 then break
40   inc A
50 until
60 'hier geht es weiter nach break
```

### 2.) `break` Zeilen-Nummer | Adresse | Label

`break` Zeilen-Nummer oder `break` Adresse oder auch `break` Label verlässt alle aktiven Schleifen auf einmal, es sei denn, ein `gosub` oder `call` ist "dazwischen". Zeilen-Nummer, Adresse oder Label sollten außerhalb des gesamten Schleifen-Blocks inklusive aller übergeordneten Schleifen liegen:

```
ex.:
10 let A = 0
20 do
30   if A>5 then break 70
40   inc A
50 until
60 'hier steht irgendeine Anweisung
70 'hier geht es weiter wenn A=6
```

Zu [Labels](#) oder [Adressen](#) siehe auch [goto](#), [gosub](#) und [Operator @](#) und [->](#).

```
ex.:
10 let A = 0
20 do
30   if A>5 then break "Weiter"
40   inc A
50 until
60 'hier steht irgendeine Anweisung
70 proc Weiter
80 'hier geht es weiter wenn A=6
```

## case

Bestandteil des `select / case` Programm-Schalters

`case` setzt die Programmausführung in Abhängigkeit von Variablenwerten fort. `select / case` wird eingesetzt, wenn es mit `if` zu komplex wird – insbesondere wenn mehrere Entscheidungen von einer Komponente abhängen. Siehe [select](#) für die ausführliche Beschreibung.

call, rcall, lcall

call Prozedurnamen(Arg1, Arg2, Arg3,...)

Arg1 ...Arg16 sind die Argumente (Werte), die an die sub-Routine übergeben werden.

call startet eine User-Prozedur, ein Unterprogramm, das eine kleine Teilaufgabe löst (ähnlich den Funktionen). Die Teilaufgabe sollte mehrmals im Programm benötigt werden, sonst lohnt eine Prozedur nicht. Das eigentliche Unterprogramm startet mit dem sub-Label:

```
ex.:
10 call Test(11.5, 2*pi, "Basic")      Aufruf der Prozedur
                                         mit 3 Argumenten

20 end.                                Hauptprogramm-Ende

...
600 sub Test(A, B, A$)                  Hier beginnt der Prozedur Block
610   print "Argumente: ";
620   print A, B, A$                     Verarbeitung der Argumente
630 end                                  Prozedur-Ende
```

Eine User-Prozedur kann weitere Prozeduren aufrufen, auch Unterprogramme mit gosub \*).

Die Variablen (A – J) und die Stringvariablen A\$ und B\$ sind lokal und nur in der Prozedur selbst sichtbar. Sie sind unabhängig von den globalen Variablen A-J sowie A\$ und B\$.

Prozeduren haben auch einen eigenen Lesezeiger für read. Nützlich, wenn die Prozedur eigene Konstanten einlesen muss.

Die Werte der Argumenteliste von call() werden eins zu eins in die Variabellenliste - in der Klammer von sub() - übertragen. Die Anzahl der Argumente (max.16) muss nicht mit der Anzahl der Variablen übereinstimmen. 'Überzählige' Variablen erhalten den Wert 0, überzählige Argumente werden nicht übertragen. Die Variabellenliste darf auch leer sein, dann auch ohne Klammer.

Prozeduren liefern keine Resultate wie die Funktionen. Zum Austausch mit dem Hauptprogramm globale Variablen verwenden.

Prozeduren enden mit end, sie sind eigenständige Programme und keine Verzweigungen (wie gosub).

end muss den Prozedurblock formal beenden, kann aber zusätzlich wie ein Ausgang aus der Prozedur benutzt werden, auch wenn der Ausstieg innerhalb von Schleifen, If- oder Select-Blöcken oder Unterprogrammen erfolgt. Noch schöner geht's mit exit.

\*) Variablen A-J sowie A\$ und B\$

Unterprogramme mit gosub, in die *innerhalb* von User-Prozeduren oder -Funktionen verzweigt wird, arbeiten mit den lokalen Variablen der aufrufenden Prozedur / Funktion.

### Rekursion

Eine User-Prozedur sollte sich nicht selbst aufrufen. Man müsste sonst bei jedem Selbstaufzug den Taskzähler überprüfen, ob er noch nicht 10 erreicht hat. Abfragen mit sys(0). Mehr als 9 User-Prozeduren/-Funktionen kann man z.Z. nicht verschachteln.

### Interrupts

Interrupt-Service-Routinen sind bereits Prozeduren. Es ist nicht nötig, weitere Prozeduren innerhalb von Interrupts zu starten.

### Funktionen aufrufen

Seit V105A30 kann **call** auch Funktionen vom TypII aufrufen (wenn man das Funktionsergebnis nicht benötigt).

statt:	geht jetzt auch:
10 <b>let A = fn Fmod(90,3)</b>	10 <b>call Fmod(90,3)</b>

**rcall** funktioniert wie **call**, aber für Prozeduren im internen **ROM**, die von Programmen im RAM aufgerufen werden sollen aber auch umgekehrt (**rcall** nicht mit F401).

Ähnliches gilt für **Icall**. **Icall** funktioniert wie **call** - aber: **Icall Test** lädt die Datei **Test.PBAS** von der SD-Card und startet die Prozedur **Test**. **Icall / sub** dürfen keine Parameter-Klammern haben. Die Datei **Test.PBAS** muss eine Prozedur Namens **Test** enthalten. **Icall** soll im **ROM** aufgerufen werden, die Prozedur selbst (**sub Test...end**) befindet sich nach dem Laden im RAM. Nach Beendigung verschwindet die Prozedur wieder aus dem RAM. Im RAM sollte kein Programm sein (wird gelöscht).

Diese Anweisung ist als Möglichkeit gedacht, im ROM eine Art Betriebssystem vorzuhalten, das bei Bedarf Kommandos oder andere Programmteile nachlädt und daher unbegrenzten Speicherplatz zur Verfügung hat.

**call, Icall, rcall** können jetzt auch im Kommando-Modus aufgerufen werden.

## cd

(change Directory, (change Drive nur F407ZG oder F405RG Boards))

**cd** oder **cd1** oder **cd2**

**cd..**

**cd\**

**cd "BasicProgs"**

wechselt in ein anderes Verzeichnis auf der SD-Karte. **cd..** wechselt in das übergeordnete Verzeichnis. **cd\** wechselt ins Rootverzeichnis.

**cd "BasicProgs"** wechselt in das Verzeichnis **BasicProgs**, welches sich im aktuellen Verzeichnis befinden muss.

**cd, cd1** oder **cd2** lesen die SD-Karten neu ein, z.B. nach Kartenwechsel. Achtung! Alle Dateien sollten geschlossen sein, sonst droht Datenverlust.

**cd2** nur für die STM32F407ZG oder F405RG Boards, die eine zusätzliche CD-Card unterstützen.

Seit V1.07 kann auch ein Pfad angegeben werden. z.B.: **cd "Udisk:\Basic\Test\"**

Der Pfad startet immer im Root-Verzeichnis. Der letzte Backslash darf nicht fehlen. Siehe auch: **path\$**. **cd1** wechselt nach Zurückstecken der SD-Card in das zuletzt eingestellte Verzeichnis, **cd** ins Rootverzeichnis. Auf dem F407ZG oder F405RG wechselt man mit **cd2** und **cd1** das Laufwerk.

## celse

Bestandteil der **select / case** Struktur. Siehe [select](#).

clear / cL

**clear [Auswahl]**

Kommando- und Programm-Modus.

<b>clear</b>	löscht alle Variablen, LEDs, Arrays, Strings, Namensvariablen
<b>clear 0</b>	wie <b>clear</b> löscht zusätzlich die speziellen Arrays <b>S()</b> , <b>T()</b> , <b>Z()</b>
<b>clear 2</b>	führt einen CPU-Warmstart aus
<b>clear 10</b>	löscht den FlashROM-Sector <b>10</b>
<b>clear A</b>	löscht nur das Array <b>A()</b> , die Arrays <b>B()...J()</b> werden analog gelöscht.
<b>clear K</b>	löscht das Array <b>K()</b> , also die Arrays <b>A()...J()</b> auf einmal
<b>clear Z</b>	löscht nur das Array <b>Z()</b> (im batterie-gepufferten RAM)
<b>clear S</b>	löscht nur das Sound Array <b>S()</b>
<b>clear T</b>	löscht nur das Text-Array <b>T()</b>
<b>clear L</b>	löscht nur das LED Array <b>L()</b>
<b>clear buf0</b>	löscht den Konsolen-Empfangspuffer
<b>clear buf1</b>	löscht den Empfangspuffer der seriellen Schnittstelle Nr.1
<b>clear buf2</b>	löscht den Empfangspuffer der seriellen Schnittstelle Nr.2
<b>clear buf3</b>	löscht den Empfangspuffer der seriellen Schnittstelle Nr.3
<b>clear buf4</b>	löscht den Empfangspuffer der seriellen Schnittstelle Nr.4
<b>clear buf5</b>	löscht den Empfangspuffer der <u>USB-Device</u> -Schnittstelle Nr.5
<b>clear buf9</b>	löscht den Empfangspuffer der <u>USB-Host</u> -Schnittstelle Nr.9
<b>clear clock</b>	löscht die Uhrenalarme
<b>clear i2c</b>	löscht alle gespeicherten I <sup>2</sup> C-Sensoren am <u>Sensorbus</u>

close

**close # Filehandler1, Filehandler2, Filehandler3, ...**      Filehandler=10...24

Schliesst eine (oder mehrere) sequentielle Daten-Dateien. Alle Zeiger werden zurückgesetzt, die Filehandler werden wieder freigegeben. Die Dateilängen werden aktualisiert. Siehe fn Open() für die Beschreibung.

cls

**cls (ohne Parameter)** löscht den Terminalbildschirm- oder LCD-Inhalt, abhängig von der Umleitung mit der Anweisung lcd. Auch im Direkt-Modus. Das Terminal muss VT100 Kommandos verstehen.

<b>cls 0</b>	löscht nur das Terminalfenster.
<b>cls 1 [, hfarb, vfarb]</b>	löscht nur das TFT-LCD mit der Farbe <b>hfarb</b> siehe <u>Farbpalette</u> und stellt die Vordergrundfarbe <b>vfarb</b> ein.
<b>cls 9</b>	löscht das TFT-LCD mit Hintergrund schwarz und Vordergrund weiss (für Konsolen-Eingabe).

**ex. :**

```
10 cls  
20 cls 1, 7
```

config / cO

Ändert verschiedene Einstellungen, Hardware, Software dauerhaft

## config clock

**Schaltet die automatische Sommerzeit/Winterzeit-Umschaltung aus oder ein.**

config clock = on        einschalten Default  
config clock = off      ausschalten

Damit die automatische Umschaltung aktiv wird, muss danach die Uhrzeit gestellt werden.

config com

Ändert die Einstellungen der Seriellen Schnittstellen Nr.1 ...4 dauerhaft:

config com=Interface-Nr, Baud, DataBits, Parität, Stop-Bits

**config com=1, 115200** Schnittstelle Nr.1, 115200 Baud  
nur Baud (Bits/Sekunde) einstellen  
8 DataBits, keine Parität, 1 Stopbit

**config com=2, 9600,9,1,1** Schnittstelle Nr.2, 9600 Baud,  
9 DataBits, ungerade Parität,1Stop-  
Bit - siehe unten

**config com=4, 2400** Fehlernachricht, wenn I2C2 eingeschaltet ist, Werte werden aber gesetzt.

Bitte folgende Werte an der jeweiligen Position einsetzen:

8 DataBits=8, 9 DataBits=9  
keine Parität=0, ungerade Parität=1, gerade Parität=2  
1 Stop-Bit=1, 2 Stop-Bits=2, 1½ Stop-Bits=15, ½ Stop-Bit=5

Abgefragt werden können die Einstellungen mit:

	Baud	D_Bits	Parity	Stop-Bits
Schnittstelle 1:	sys(41)	nicht einstellbar ...		
Schnittstelle 2:	sys(42)	sys(142)	sys(242)	sys(342)
Schnittstelle 3:	sys(43)	sys(143)	sys(243)	sys(343)
Schnittstelle 4:	sys(44)	sys(144)	sys(244)	sys(344)

## config dataptr

<code>config dataptr = "b"</code>	8Bit-Byte	ohne VZ
<code>config dataptr = "w"</code>	16Bit-Word	ohne VZ
<code>config dataptr = "l"</code>	32Bit-Long	ohne VZ
<code>config dataptr = "f"</code>	64Bit-Float	mit VZ

Ändert das Default Pointerziel des Operators **->**, wenn die Adresse dem Basic nicht bekannt ist. Basic kennt die Adressen, die man mit **@** ermitteln kann.

Hat nichts mit dem `dataptr` von `read` zu tun, `config` lehnt sich hier nur das Token aus. VZ = Vorzeichen:

```
config dataptr = "w"          Zeiger auf 16Bit
let A = 0x40020C14            Adresse von relay
let ->A = 8                  Wert an relay
? ->A                         Testausdruck
```

`config diginp = Wert1 [, Wert2 ]`

Ändert die Beschaltung der Pullup-/Pulldown-Widerstände an den digitalen Eingängen nach dem Reset. **Wert1** ist ein 12-Bit Wert, bei dem jedes Bit einem Eingang entspricht. Ist das Bit **1** dann Pullup. Ist das Bit **=0** dann Pulldown:

`config diginp = %1011` schaltet die Pullups der Eingänge 1,2 und 4 ein, alle anderen Eingänge haben Pulldowns

Die Eingänge 1...5 können on-Interrupts auslösen. Werden sie mit mechanischen Kontakten beschaltet, ist der Totzeit-Zähler hilfreich, der die mehrfache Auslösung der Interrupts durch das Prellen der Kontakte unterdrückt. **Wert2** schaltet den Timer für jeden der 5 Eingänge (Bits 0...4) individuell ein. Ist das entsprechende Bit in **Wert2** gesetzt, wird der zugehörige Timer aktiviert:

`config diginp = 0, %1101` schaltet alle Pulldowns ein, sowie die Totzeit-Timer der Eingänge 1, 3 und 4 ein

Die Timer-Einstellung wird sofort aktiv. Der Zustand der Konfiguration lässt sich auch abfragen:

Pullups/Pulldowns mit: `sys(32)`  
Totzeit-Timer mit: `sys(33)`

`config fix = 3`  
`config fix = 2`

Ändert die Stellenzahl der Funktion fix auf 3 Stellen  
Ändert die Stellenzahl der Funktion fix auf **2 Stellen**

`config gps = on`  
`config gps = off`

Schaltet GPS-Parsing ein.  
Schaltet GPS-Parsing aus. **Default**

Um eine günstige GPS-Antennenposition zu finden (mit der Anweisung `gps`), darf das GPS-Parsing noch nicht eingeschaltet sein. Die Anweisungen `get #3` und `input #3` funktionieren mit eingeschaltetem GPS nicht. Einstellungen am GPS-Empfänger kann man mit `print #3` oder `put #3` trotzdem senden.

`config i2c = off`

Schaltet das Scannen des Sensorbusses aus, dadurch verkürzt sich die Hochfahrzeit. Ändert sich die Beschaltung *nicht mehr* (oder werden nur Sensoren angeschlossen, die vom System nicht besonders unterstützt werden), kann auf das Scannen verzichtet werden, die erkannten Sensoren wurden dauerhaft gespeichert. I2C-Sensoren, die extra gestartet werden müssen, werden aber trotzdem, wie vom Hersteller vorgeschrieben, gestartet. - **Default**

`config i2c = on`

Schaltet den Sensor-Scan ein. Nach dem Reset wird der Sensorbus gescannt, um die speziellen Sensor-Funktionen oder Anweisungen für die erkannten Sensoren freizugeben. Es müssen mindestens die Pullup-Widerstände angeschlossen sein, sonst hängt sich das System auf.

`config i2c2 = on` Schaltet die Serielle Schnittstelle 4 aus und stattdessen den I2C2-Bus (mit Standartgeschwindigkeit) auf die Portpins PB10/11. Resettaste drücken!

`config i2c2 = off` Schaltet die Serielle Schnittstelle 4 wieder auf die Portpins PB10/11 und den I2C2-Bus aus. – **Default** Resettaste drücken!

`config input = 1` Ändert den Eingabemodus nach Reset auf Einfügemodus

`config input = 0` ändert den Eingabemodus nach Reset auf Überschreibmodus – **Default**

`config input as Konsole` ändert die Konsolen-Schnittstelle. Der Basic-Computer startet *stets* mit Schnittstelle 1 und wechselt bei konfiguriertem USB auf die USB-Schnittstelle Nr. 5 / 9. Danach kann eine Umleitung auf ein anderes Interface vorgenommen werden, wie mit `set input` jedoch automatisch nach jedem Reset. `Konsole` kann die Werte 0...9 annehmen. Siehe [set input](#). Ist das eingestellte Interface nicht aktiv, wird die Umleitung ignoriert. [Verbosity](#) sollte ausgeschaltet werden, dann werden keine Start-Meldungen über com1, 5 oder 9 gesendet.

`config lcd = Typ-Nr.` Ändert das LCD-Display:

Typ-Nr.	Controller	Auflösung	Data
0	kein TFT-Display angeschlossen		
1	ILI9488	480x320	SPI 24Bit *)
2	ILI9341	320x240	SPI 16Bit **)
3	ST7793	480x200	SPI 16Bit
4	HX8357B	480x320	Arduino
5	HX8357C	480x320	Arduino
6	SSD1963	800x480	Parallel 16Bit
7	ST7789V	320x240	SPI 16Bit
10	schaltet die Farbpalette aus		
11	schaltet die Farbpalette ein		
12	Display nicht rotiert		
13	Display 180° rotiert		

Ist die Farbpalette ausgeschaltet, gibt es keine Farbnummern mehr, es werden stattdessen direkt die Farbwerte ([RGB565](#)) benutzt. Klappt nur mit 16Bit Display-Controllern.

\*) nur mit 3.5"-Displays getestet

\*\*) getestet mit 2,4", 2.8" und 3.2"-Displays

#### Reset erforderlich

z.Z. werden nur Typ1 und 2 unterstützt. Typen 4, 5 und 6 werden nur mit 144Pin CPU (STM32F407ZG) arbeiten. Typ-Nr. abfragen mit sys(53), Palette ein/aus mit sys(54). Typ-Nr. 10-13 sind *zusätzliche* Einstellungen.

Für das [Standart-Display](#) (407MiniBoard) muss folgendes eingestellt werden:

`config lcd = 1` LCD Controller-Type ILI9488  
`config lcd = 12` Display nicht rotiert

Dieses Display läuft mit 25MHz SPI-Frequenz, sodass mit kurzen

Verbindungen gearbeitet werden muss. Die SD-Card sollte diese Geschwindigkeit unterstützen z.B.: San Disk Ultra 32GByte.

#### config led = Anzahl, Ursprung

Ändert die Anzahl der LEDs im LED-Streifen auf **Anzahl** LEDs. **Anzahl** muss kleiner als **257** sein.

Ursprung muss nur angegeben werden, wenn es sich um ein LED-Feld mit 8x32 Neopixel-LEDS handelt. Ursprung = **1** setzt Koordinate 0,0 auf links unten.

Ursprung = **0** setzt Koordinate 0,0 auf links **oben**.

config led = **256, 1** für ein solches Panel

#### config load = 1

Erzwingt das Laden von Programm-Datei **0** aus dem internen ROM.

#### config load = 0

stellt den Standard wieder her (Laden der Programm-Datei **0** aus dem seriellen Flash).

#### config morse = relay

Stellt die Ausgabe der Anweisung **morse** auf das Relais Nr.**16** um.

#### config morse = sound

Stellt die Ausgabe der Anweisung **morse** wieder auf den Soundausgang um. - **Default**

#### config peek

Ändert die Anzahl Bytes, die die Funktion **peek()** einliest. Auch die Interpretation des Wertes ändert sich:

config peek “**b**” peek liest Bytes (8-Bit ohne VZ)  
config peek “**w**” peek liest Words (16-Bit ohne Vorzeichen)  
config peek “**l**” peek liest Longwords (32-Bit ohne VZ)  
config peek “**f**” peek liest Fliesskomma (64-Bit)

#### config pwm1...pwm4

Konfiguriert die PWMs:

config pwmx=Frequenz, Startwert x=1...4

```
10 config pwm1=50, 8000  
20 config pwm2= , 1500  
30 config pwm3= , 4000
```

stellt für z.B. **pwm1**: **50Hz** und Startwert **8000** ein.

**pwm1...pwm4** teilen sich einen Timer (Timer5), sodass die Frequenz für alle 4 Kanäle gilt. **Default**-Werte sind: 1000Hz, 50%.

Abhängig von der Frequenz berechnet das System die Auflösung. Sie kann mit **sys(14)** abgefragt werden. Dieser Wert ist auch der maximale Startwert (=100%) für **pwm1 ... pwm4**. **pwm4** gibt es nur auf F401/F411, F405RG und F407ZG.

**pwm1** auf **100Hz** und **15%** Startwert einstellen:

```
10 config pwm1=100, sys(14)*15/100
```

`config relay = Wert1 [, Wert2 ]`

Ändert den Zustand der 'Relais'-Ausgänge nach dem Reset. `Wert1` ist ein 16-Bit Wert, bei dem jedes Bit einem Ausgangspin entspricht. Ist das Bit = **1** dann ist der zugehörige Pin nach dem Reset direkt eingeschaltet. Ist das Bit = **0** dann bleibt er aus.

`config relay = %10111` schaltet Relais 1,2,3 und 5 ein

Je nachdem wie die externe Beschaltung aussieht, kann es auch sinnvoll sein, die Ausgänge als OpenDrain zu betreiben (z.B. wenn **5V** geschaltet werden sollen)

Dafür gibt es `Wert2`, der für jedes Bit den zugehörigen Ausgang als OpenDrain einstellt:

`config relay = %1111, %1001`

schaltet die Ausgänge `relay 1...4 ein` sowie `relay 1 und relay 4 als OpenDrain`, für jedes relay-Bit ein entsprechendes OpenDrain-Bit:

`config relay = 0, 65535`

schaltet sämtliche digitalen (Relais) Ausgänge **aus** und als OpenDrains. Setzt man das entsprechende Bit auf Null, ist der zugehörige Ausgang wieder im Gegentaktmodus.

Der Zustand der Konfiguration lässt sich auch abfragen:

Zustand der Ausgänge nach Reset: `sys(30)`  
OpenDrain-Zustand nach Reset: `sys(31)`

`config save = 1`

Erzwingt das Speichern von Programm-Datei **0** ins interne ROM.

`config save = 0`

stellt den Standart wieder her (Speichern der Programm-Datei **0** ins serielle Flash).

`config set = false`

Für Relaiskarten mit negativer Logik (**0=On, 1=Off**) reicht es nicht aus, nur die Relais nach einem Reset auf Logikpegel **high** zu setzen (mit `config relay`). Die Anweisungen `set relay` und `reset relay` sollten sinnvollerweise *invers* arbeiten. Das geschieht mit dieser Einstellung. Die System-Variable `relay` ist davon selbst nicht berührt. Bei negativer Logik schaltet ein **0**-Bit halt ein: `set relay1` oder aber: `let relay.0 = 0` und ein **1**-Bit eben aus.

`config set = true`

Stellt wieder positive Relais-Logik her. – Default

`config start v=0`

Verringert die Startup-Meldungen auf ein Minimum.

`config start v=1`

Es werden sämtliche Startup-Meldungen angezeigt.

`config start l=0`

USB-Keyboard mit deutscher Tastenbelegung. Das Datum wird deutsch angezeigt: Di, 24.12.2024

`config start l=1`

USB-Keyboard mit US-amerikanischer Tastenbelegung. Das Datum ist US-englisch: Tue, 12.24.2024

<code>config start r=0</code>	Der ROM-Speicher für BASIC ist 128 KB gross und belegt Sector 11. <b>Default</b>
<code>config start r=1</code>	Der ROM-Speicher für BASIC ist 256 KB gross und belegt Sector 10 & 11. Keine Maschinenprogramme! Aber Vorsicht, nach der Umstellung sind im ROM bereits vorhandene Programme nicht mehr lauffähig. Sie müssen neu ins ROM geladen werden. Programme >64K können nur mit <code>rload</code> gespeichert werden. Wird neue Firmware geflasht, wird der Sektor 10 gelöscht, Sektor 11 hingegen nicht!
<code>config usb = 0</code>	<b>Schaltet den USB-Anschluss aus.</b> Die Konsole wird auf Schnittstelle 1 gestartet.
<code>config usb = 1</code>	Schaltet den USB-Anschluss ein – wird wie eine 5. serielle Schnittstelle bedient.
<code>config usb = 2</code>	<b>Schaltet den USB-Anschluss ein und leitet Konsolen-Ein-/Ausgaben auf den USB um.</b> – <b>Default</b> Erkennt der Basic-Computer beim Hochfahren keine aktive USB-Schnittstelle, wird die <u>serielle Konsole</u> gestartet.
<code>config usb = 3</code>	<b>Schaltet den USB-Anschluss im Host-Modus ein.</b> Es kann eine USB-Tastatur angeschlossen werden. Damit der USB-Host-Anschluss ordnungsgemäss funktioniert, muss eine kleine <u>Zusatplatine</u> an die USB-Portpins auf dem Board angeschlossen werden. Das Keyboard ist als Schnittstelle #9 auslesbar. <code>get #9</code> und <code>input #9</code> dienen diesem Zweck. Die Konsole wird auf Schnittstelle 1 verschoben. Dort sollte auch ein Konsolengerät angeschlossen sein, sonst sperrt man sich aus.
<code>config usb = 4</code>	<b>Startet die Konsole auf der Kombination von USB-Tastatur und TFT-LCD (Interface Nr.9).</b> Da die SPI-TFTs, die am 407Mini-Board anschliessbar sind, sehr klein sind und nicht scrollen können, ist ein vernünftiges Arbeiten auf dieser Kombi nicht wirklich möglich. Es ist vielmehr ein Experiment für die zukünftige Entwicklung eines eigenständigen Basic-Computers. Für die Eingabe von Kommandos und Mini-Programmen reicht es aber schon jetzt.  Bei Displays *), die nicht scrollen können, wird das Display bis zur untersten Zeile beschrieben. Dann wird die Ausgabe gestoppt und auf einen Tastendruck gewartet. Nach der Tastenbetätigung wird der Screen gelöscht und die Ausgabe in der obersten Zeile fortgesetzt. Gibt man ein Kommando in der untersten Zeile ein, kann es deshalb sein, dass man das Ergebnis nicht mehr sieht. Wenn der Platz auf dem TFT knapp wird, <code>cls</code> für einen leeren Screen ausführen! TFTs mit Hardware Scrolling benötigen diese Vorgehensweise nicht. Auf diesen TFTs arbeitet man

wie mit der Terminal-Emulation auf dem PC. Sollen Verzeichnisse oder Listings angezeigt werden, die nicht vollständig aufs Display passen, hält die Anzeige aber in jedem Fall an und wartet auf einen Tastendruck (es gibt ja keinen Text-Pufferspeicher wie in TeraTerm).

\*) Diese Displays verhindern u.U. die unbeaufsichtigte Autostartfunktion, weil sie nicht alle Startmeldungen darstellen können. Verbosity abschalten schafft Abhilfe:

```
config start v=0
```

Die USB-Einstellungen werden nach dem nächsten Reset wirksam. Siehe auch Anweisung [usb](#).

Die mit `config` geänderten Einstellungen werden im batterie-gepufferten RAM (RTC\_Backup\_Register) dauerhaft gespeichert, es muss daher eine 3V Uhrenbatterie (CR2016, CR2025, CR2032 evtl. mit Batteriehalter) angeschlossen sein. Einige Einstellung werden erst nach Reset wirksam. Alle Einstellungen können mit `reset config` wieder auf den Auslieferungszustand (**Default**) zurückgesetzt werden. Hat man versehentlich eine Einstellung vorgenommen, die den Zugriff auf das System unmöglich macht, hilft es, die Uhrenbatterie zu entfernen und die Stromversorgung auszuschalten. Die Einstellungen sind dann alle futsch. ☺

**Default**-Werte sind Werte, die vom System angenommen werden, wenn (noch) keine Angaben gemacht wurden. (default = Unterlassung)

Verbosity=Geschwätzigkeit, Verbosity=on: sämtliche System-Meldungen werden angezeigt, Verbosity=off: keine oder nur die nötigsten Meldungen.

cont

(continue)

`cont` kann sowohl im Programm- als auch im Direktmodus (Kommando-Modus) eingesetzt werden. Abhängig vom Modus hat `cont` zwei verschiedene Aufgaben:

### 1.) im Kommando-Modus

```
> cont
```

Im Kommando-Modus setzt `cont` die Programmausführung nach Programmabbruch (mit `ctrl c`) fort. Nach einem Fehler ist die Fortsetzung mit `cont` nicht möglich.

### 2.) im Programm

```
ex. :  
10 for N=1 to 10  
20     if N=5 then cont  
30     print N  
40 next N
```

Im Programm überspringt `cont` die Programmausführung innerhalb von Schleifen bis zur zugehörigen `next`-, `until`- oder `wend`-Anweisung, ohne die Schleife zu verlassen. Es ersetzt hier z.B. eine `goto 40` Anweisung. In diesem Beispiel bewirkt es, dass die Zahl 5 nicht ausgedruckt wird.

convert / conV

startet eine Messwandlung auf DS18x20 Temperatursensor. Siehe [sensor\(\)](#)

`convert Sensor-Serien-Nummer`  
`convert 6`  
`convert 7`

Die **Seriennummer** des auszulesenden Sensors im [One-Wire-Bus](#) (an Portpin PA6 = Pin 28 auf Header1 (J3) des 407-MiniBoards) steht entweder in einer **Stringvariablen** oder einem **Stringliteral** in Anführungszeichen.

Wie man an die **Serien-Nummer** kommt, ist [hier](#) beschrieben.

```
ex. :  
10 let A$ = "6630167858422087464"                                Seriennummer-(Variable)  
20 convert A$  
30 wait 1000  
40 print sensor(A$)  
  
ex. :  
10 convert "6630167858422087464"                                Seriennummer-(Literal)  
20 wait 1000  
20 print sensor("6630167858422087464")
```

`convert 6` startet eine Messung eines einzelnen Sensors an Portpin PA6, Pin 28 auf Header1 (J3), dessen Serien-Nummer man nicht kennen muss.

`convert 7` startet eine Messung eines einzelnen Sensors an Portpin PA7, Pin 27 auf Header1 (J3), dessen Serien-Nummer man nicht kennen muss:

```
ex. :
10 convert 6
20 wait 1000
30 print sensor(6)
```

## data

Programm-Daten. Mit `data` ist es möglich, konstante Werte oder Texte im Programmtext unterzubringen. `data` steht immer am Zeilenanfang. Die Daten werden durch Kommata getrennt. Es können so viele Daten in eine Zeile gepackt werden, wie hineinpassen.

```
ex. :
1000 data 0.998, 0.986, 0.977, 0.944, 0.932, %10101, 0x1AF
1010 data "sinustab", "rechtecktab", "sägezahntab"
```

Mit `read` werden sie gelesen. `data`-Zeilen dürfen keinen Programm-Text enthalten und sollten zusammenhängend im Block und am Besten am Programm-Ende angeordnet sein. Numerische Daten dürfen nur Zahlen sein. Text-Daten sind nur als Literale in Anführungszeichen möglich. Man kann aber auch nicht-druckbare (oder nicht eintippbare) Zeichen einbauen, indem man die Zeichenfolge "`\nxyz`" oder "`\xab`" einfügt, z.B:

```
1000 data "pahlbasic\n013\n010ist super"
```

Test-Kommandos:

```
> read A$
ready
> print A$
pahlbasic
ist super
ready
```

`xyz` ist die 3-stellige (!) ASCII-Nummer des einzufügenden Zeichens. Das Beispiel fügt eine Zeilenschaltung ein. Wem 3 Stellen zu lang sind, kann auch Hex-Zahlen (2-stellig) einsetzen. Statt `n` schreibt man `x`:

```
1000 data "pahlbasic\x0D\x0Aist super"
```

## debug / deB

`debug on` schaltet den Debug-Modus ein und `debug off` schaltet ihn aus. Nach einem Reset ist der Debug-Modus standardmäßig eingeschaltet, d.h. Programmabbruch (mit `Strg + c`) und Einzelschritt-Modus (mit `Strg + d`) sind erlaubt. Nach der Ausführung von `debug off` sind `Strg + c` und `Strg + d` gesperrt.

Dies ist evtl. sinnvoll, wenn der Basic-Computer als autonomes Gerät eingesetzt wird und auch Daten über RS232 empfängt - kann aber zu Problemen führen, wenn die Autostartfunktion (mit `save 1`) aktiviert wurde, ein Anhalten des Basic-Computers ist dann nur noch möglich, wenn ein *Programmabbruch* im Programm *eingebaut* wurde. Das gilt verstärkt, wenn auch noch mit `on err` der Programmabbruch bei Fehlern abgeschaltet wurde. Siehe auch [end](#).

Nach Einschalten des Einzelschritt-Modus (mit `Strg + d`) hält das Programm an, bis die Leertaste gedrückt wird. Es wird die jeweils nächste Instruktion auf dem Terminal gelistet und mit dem Tastendruck abgearbeitet. Instruktionen, die selbst eine Eingabe vom Terminal erwarten, arbeiten evtl. nicht korrekt.

`dec, decr`

\*

`dec Variable [, Wert]`

**ex.:**

`dec D(9)`

`dec` Verringert den Wert der Variablen oder des Array-Elements (hier: `D(9)`) um 1. Programm und Direkt-Modus. (siehe auch `inc`) Gibt man den Wert zusätzlich an, wird um diesen Wert verringert.

`Wert` ist eine Ganzzahl > 0 und < 32768. Das Ergebnis wird nicht geprüft.

\*

`decr (System-)Variable 1, (System-) Variable 2, ...`

**ex.:**

`decr N, M, font`

`decr` kann zusätzlich auch Systemvariablen sowie das Array `Z()` und auch gleich mehrere Elemente auf einmal. (`decr` ist ca. 10% langsamer als `dec`). Es wird stets um 1 verringert.

`deffn`

erklärt eine User-definierte-Funktion:

`deffn Funktionsname() as Funktions-Nummer` nur für Funktionen vom Typ 1

**ex.:**

`10 deffn Test() as 1`

`deffn` meldet die User-Funktion `Funktionsname` an. `deffn` ermittelt die Adresse des zugehörigen Programmzeilen-Blocks, der mit dem Label `function Funktionsname` beginnt und verknüpft sie mit der `Funktions-Nummer`.

Es gibt 5 User-Funktionen vom Typ1. Siehe User-definierte-Funktionen1 und Userdefinierte Funktionen2 für eine ausführliche Beschreibung. User-Funktionen vom Typ 2 benötigen keine vorherige Definition mit `deffn`, dafür starten sie etwas langsamer.

User-Funktionen vom Typ 1 bestehen aus 3 Teilen: 1.) Definition mit `deffn`, 2.) Aufruf mit `fn1()...fn5()` 3.) Programmzeilenblock mit den Anweisungen, der mit dem Label function Funktionsname beginnt.

`del`

`del "Name"`  
`del Nummer`  
`del !`  
`del @`

`del` löscht das Programm mit dem Namen `Name` oder der Nummer `Nummer` auf dem Flash-Chip. Die Datei-`Nummer` kann mit `dir` oder `fn Dir()` ermittelt werden. Wenn `Nummer=32` wird der gesamte Flash-Chip (2Mbyte\*) gelöscht. `del 0` löscht die versteckte Datei '0000'.

`del !` oder `del @` löscht den ROM-Speicherplatz für Programme bzw. Einstellungen (128KB/256KB).

Bevor gelöscht wird, muss man eine Sicherheitsabfrage mit "y" beantworten (Rückfrage nur im Kommando-Modus):

```
> del "speedtest"
File Nbr.: 29 Will Be Deleted!!
overwrite?
```

\*) Bei 4MByte EEPROM: **del 64** für ganzen Chip, bei 8MByte Flash: **del 128**

**dir**

(Directory)

**dir [Medium]**

Medium: 1=Flash, 2=SD-Card

Druckt das Verzeichnis des seriellen Flash-Speichers oder der SD-Karte auf dem Terminal aus. Auf dem Flash: Die Dateien werden mit Namen und laufender Nummer ausgedruckt. pahlbasic hat ein sehr simples „Datei-System“. Es gibt 32 „Dateien“ (Slots 0-31) - jede ist 64Kbyte groß \*).

Die Datei Nr.0 wird nicht angezeigt (Programm-Datei Nr.0 wird bei Start des Basic-Computers automatisch geladen):

```
> dir 1

-----directory-----
1 soundtest.....prg    69 Bytes
2 stringtest.....prg    71 Bytes
3 string_test.....prg   156 Bytes
4 speed_test.....prg   41 Bytes
5 rem+@+select-test...prg 206 Bytes
6 if-test.....prg    124 Bytes
7 data_test.....prg   191 Bytes
8 ON-test.....prg    44 Bytes
9 Apfeliaennchen.....prg 597 Bytes

ready
```

Anzeige: Directory auf dem FlashROM

Auf der SD-Karte wird ein DOS-ähnliches Verzeichnis angezeigt:

```
> dir 2
VOLUME:\ArmBasic32\
---D---      DIR 03/12/2023 15:51 .
---D---      DIR 03/12/2023 15:51 ..
--A---- 6681348 02/12/2023 17:25 armbasic.pdf
--A---- 1268007 03/12/2023 09:43 ArmBasicV105A31e4.hex
--A---- 6577 23/11/2023 10:56 pahlbasic_bright.xml
--A---- 6612 23/11/2023 10:56 pahlbasic_dark.xml
--A---- 1928 27/11/2023 10:43 readme.txt
--A---- 4280 18/11/2023 18:05 SD-Card-Errors.txt
--A---- 204800 21/08/2021 09:58 TFTcolour.exe
---D---      DIR 05/11/2023 07:18 BasicExamples
---D---      DIR 05/11/2023 07:18 Sound-Examples

ready
>
```

Anzeige: Directory auf der SD-Card

\*) Gilt für den Standard Chip (2MByte), auf den 4MByte EEPROM passen 64 Dateien und auf dem 8MByte Flash-Chip lassen sich 128 Dateien speichern.

do

Einleitung einer Programmschleife. Abgeschlossen wird die Schleife mit [until](#):

Der Anweisungsblock zwischen `do` und `until` wird wiederholt *bis* der Ausdruck nach `until` wahr ( $<>0$ ) *wird*:

**ex.:**

```
10 let time = 0
20 do
30   inc A
40 until time=10
50 print A
```

als Programm-Block

oder in einer Zeile:

```
10 let time = 0 : do : inc A : until time=10 : print A
```

Mehrere (max. 20) `do`-Schleifen können ineinander verschachtelt werden. `do` wird gerne verwendet, wenn auf die Erfüllung einer Bedingung gewartet werden muss (z.B. ein Eingang einen bestimmten Wert annimmt). Reine Zählschleifen gehen mit `for` / `next` besser. `do`-Schleifen werden mindestens einmal durchlaufen. Eine ewige Schleife erhält man, wenn die Bedingung nach `until` weggelassen wird.

draw

Zeichnen von einfachen graphischen Elementen auf das TFT-Display

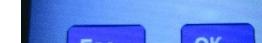
**draw "a"** zeigt einen Bildschirm für die Kalibrierung der Touch-Funktion, es müssen 4 Punkte angetippt werden, nach Anweisung auf dem TFT. Eventuell wird das Display rotiert und das System neu gestartet.

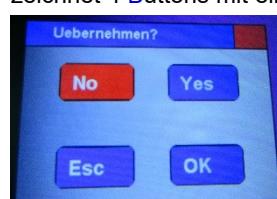
`draw "b4", A` zeichnet 4 Buttons, die frei mit Text belegbar sind. Die Texte der Buttons stehen in den Strings `A$...D$`. Es können ein Titel und ein Hinweistext in den Strings `G$` und `H$` übergeben werden. Klicken auf das rote Quadrat, dient dem Verlassen der Box. Ergebnis: Bits 0...3 von `A`.

**draw "b1", A** zeichnet 4 Buttons, die frei mit Text belegbar sind. Wie „**b4**“.  
„**b4**“.  
 Die Buttons lösen sich aber gegenseitig aus. (Radio-Buttons)



draw "by", A  
zeichnet 4 Buttons mit einer Yes / No Auswahl sowie einen OK-Button und einem Escape-Button und speichert die getroffene Auswahl in der Variablen A: No=1, Yes=2. Mit OK=A+4 oder Esc=27 verlässt man die Box. Der FrageText steht in H\$.





**draw "br", A**

zeichnet 4 Buttons, die direkt mit den eingebauten 4 Relais verbunden sind. Klicken schaltet die Relais. Das Ergebnis wird zusätzlich in der Variablen hier: **A** gespeichert, d.h. es werden die 4 unteren Bits von **A** gesetzt. Klicken auf das rote Quadrat schließt die Box.



**draw "c", x, y, Radius**

zeichnet einen Kreis (Circle) um den Punkt **x, y** mit **Radius**.

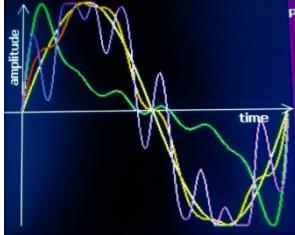
**draw "f", uf, ff, ss, gf**

stellt die Zeichenfarben ein. Ist die Farbpalette eingeschaltet, Farbnummern angeben, sonst direkt die Farbwerte:

**uf** = Umrißfarbe, **ff** = Füllfarbe, **ss** = Strichstärke (in Pixel),  
**gf=0** ungefüllt, **gf=1** gefüllt, **gf=2** gefüllt mit Verlauf  
(siehe [draw "v"](#)) **gf=3** gefüllt, Füllfarbe wird zur Hintergrundfarbe z.B. für Beschriftung mit [text](#).

**draw "g"**

Zeichne Graph:



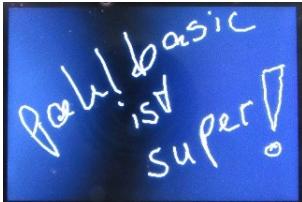
Stellt den Inhalt des Sound-Arrays **S()** als Schwingungskurve 'Oszilloskopogramm' dar. Im Bild links sind 4 Kurven übereinander abgebildet. Sinuskurve (gelb), Fourierkurve (grün), sowie 2 FM-Kurven orange und violet.

**draw "g", 0** zeichnet Koordinatenkreuz + gelben Graph  
**draw "g", 1** zeichnet nur den grünen Graph  
**draw "g", 2** zeichnet nur den orangenen Graph  
**draw "g", 3** zeichnet nur den violetten Graph

```
10 set sound : draw "g"  
20 call Fourier(5,4,3,2,1,1) : draw "g", 1  
30 call Fmod(30,5) : draw "g", 2  
40 call Fmod(90,7) : draw "g", 3
```

**draw "h"**

auf diesem Testschirm kann man mit einem Stift zeichnen.  
(Testet die Kalibrierung).  
Abbruch der Funktion: eine Taste auf dem Terminal betätigen.



**draw "l",x1,y1, x2,y2**

zeichnet eine Linie von Punkt A (**x1,y1**) zum Punkt B (**x2,y2**).

`draw "m"`

zeichnet eine **Message-Box** auf das TFT und wartet auf Anklicken. Die „Message“ muss in **H\$** stehen und wird auf 15 Zeichen gekürzt.



`draw "o"`

zeichnet nur den „**OK**“-Button auf den TFT und wartet auf Anklicken

`draw "p", x, y, f`

setzt ein Pixel mit den Koordinaten **x, y** und Farbnummer **f**. siehe auch [plot](#)

`draw "r", x1, y1, Breite, Höhe [,Radius]`

zeichnet ein Rechteck mit **x1, y1** linke obere Ecke, **Breite** und **Höhe**. Keine schrägen Rechtecke möglich. Ist die Angabe **Radius** vorhanden, wird ein Rechteck mit runden Ecken gezeichnet

`draw "s"`

malt den **Info-Screen** auf das TFT, der ständig aktualisiert wird, bis eine Taste auf dem Terminal gedrückt wird.



`draw "t", x1, y1, Farbe`

zeichnet ein kleines Textfeld (wie auf dem Info-Screen) an die Koordinaten **x1, y1** mit Feld**farbe**. Der Text steht in **H\$**.

`draw "v", sf, ef, rg`

stellt einen Farbverlauf für Rechtecke und Kreise ein. Funktioniert nur, wenn die Farbpalette ausgeschaltet ist, also nicht bei Displays mit dem ILI9488. Es müssen die **Farbwerte** angegeben werden:



**sf**=Anfangsfarbe, **ef**=Endfarbe (nicht die Füllfarbe!)  
(z.B.: 0=Schwarz, 65535=weiss)

**rg**=0 von oben nach unten, **rg**=1 von links nach rechts

`draw "Dateiname" [, X, Y]`

stellt die Bilddatei mit Namen "**Dateiname**" auf dem TFT LCD dar. Bilddatei muss ein Windows-Bitmap sein, mit 24Bit Farbtiefe (Standart) und sollte nicht viel grösser sein, als das Display anzeigen kann (wird nicht skaliert). Fotos kann man ja z.B. mit Gimp anpassen und als Bitmap exportieren. Ist das Bild kleiner als das Display (z.B. ein Symbol) kann man die Pixel-Position der linken oberen Ecke mit **X** und **Y** angeben. Ohne Angabe wird das Bild auf dem Display zentriert. Bitmaps sind für einen Micro-Controller ohne Rechenaufwand zu zeichnen. Sie benötigen viel Speicherplatz, aber der ist auf einer SD-Karte ja reichlich vorhanden. Ohne SD-Karte geht's nicht. Auf Displays mit ILI9488 werden die Bilder mit 24Bit Farbtiefe angezeigt, auf allen anderen nur mit 16Bit.

Ist die Konsole auf das TFT umgeleitet und `draw` wird im Kommando-Modus ausgeführt, dann wartet `draw` im Anschluss auf die Darstellung auf die Betätigung einer Taste bis wieder zum Direktmodus zurückgekehrt wird.

Für alle `draw`-Anweisungen gilt:

Die Koordinaten müssen ins Display passen (sonst fehlen Teile der gezeichneten Elemente) `x=0...479, y=0...319` für Displays mit 480x320 Pixel. (`x=0...319, y=0...239` für Displays mit 320x240 Pixeln). Farbnummern sind `0...63`, z.B.: `0=schwarz, 9=weiss` siehe Farbpalette. Bei ausgeschalteter Palette und 16-Bit RGB-Farben reichen die Farbwerte von `0=schwarz` bis `65535=weiss`. Ein Beispiel mit eingeschalteter Farbpalette:

```
ex.:
10 draw "f" 0, 6, 2, 1      Farbe: Umriss schwarz, Fläche blau, Strich: 2-Punkt,
                             gefüllt (natürlich ohne Verlauf)
                             Gilt für alle folgenden Elemente, bis zur nächsten
                             draw "f" -Anweisung.

20 draw "r" 30, 30, 100, 40, 5   Rechteck mit Koordinaten 30/30, Breite 100, Höhe 40
                                 Pixel und abgerundeten Ecken (Radius 5 Pixel)
```

## dump / dU, rdump

eine Art einfache Hilfefunktion, `dump` listet verschiedene Parameter auf:

- \* `dump 0` listet den Inhalt des Programmspeichers Byte für Byte auf
- `dump 1` listet die 23 vordefinierten Farben auf
- `dump 2` listet die 8 eingebauten Schriften auf
- `dump 4` Übersicht über die Funktion `sensor()`
- \* `dump 5` die aktuellen Werte der Analog-Eingänge, fortlaufend
- `dump 7` listet die On-Interrupts auf (im Direktmodus nur, wenn das Programm nicht mit `end.` sondern mit `end` beendet wurde)
- `dump 8` Inhalt des Konfigurations-Speichers
- `dump 9` Übersicht über die Funktion `sys()`
- \* `dump A ...J` Auflistung des gleichnamigen Arrays (alle 100 Werte)
- \* `dump S` Auflistung des Inhalts der 8 Stringvariablen (wenn nicht leer)
- \* `dump V` Auflistung der Namens-Variablen + Werte
- \* `dump "Dateiname"` listet die Datei sektorweise (als Hex-Ansicht, nur Flash-Chip)
- \* `dump X` Auflistung der geöffneten Dateien auf dem Flash Chip

\* auf F411/F401 nur diese markierten Parameter

```
> dump 8
Contents of the configuration memory:
=====
Register .Bit Value Function
----- - - -----
RTC_BKP4R     115200 Baud Konsole Speed

RTC_BKP5R     22008C36 multiple Settings:
RTC_BKP5R.0-1    2 peek(longword)
RTC_BKP5R.2-3    1 pointer = ->word
RTC_BKP5R.4      1 negative Relay-Logic
RTC_BKP5R.5      1 Insert-Mode
RTC_BKP5R.6      0 I2C Scanning off
RTC_BKP5R.7      0 automat. Summertime On and Activated
RTC_BKP5R.8      0 fix$: 2 digits
RTC_BKP5R.9      0 USB serial Device activated
RTC_BKP5R.10     1 USB Console switched off
RTC_BKP5R.11     1 USB Keyboard activated
RTC_BKP5R.12     0 Morse Output on Sound
RTC_BKP5R.13     0 LED-Coordinate 0,0 is left up
RTC_BKP5R.14     0 UART Serial_4 on PB10/11
RTC_BKP5R.24     0 use Color Palette
RTC_BKP5R.23     0 Display not rotated
RTC_BKP5R.25-27   1 Display: ILI9488 480x320 SPI
RTC_BKP5R.28     1 MIDI off
RTC_BKP5R.29     1 GPS on
RTC_BKP5R.30     0 Load from Serial Flash
RTC_BKP5R.31     0 Save to Serial Flash
```

```

RTC_CR.18          0  Wintertime aktiv

RTC_BKP6R  FFFFFFFF  Opendrain/Relays after Reset
RTC_BKP7R  00010000  Pullups/Inputs after Reset

RTC_BKP9R  80000003  Recognized I2C-Sensors: 3      * =supported
RTC_BKP9R.0    1  HTU31   *
RTC_BKP9R.1    1  BME280  *
RTC_BKP9R.31   1  24CXX

RTC_BKP10R     50  PWM-Frequency (Hz)
RTC_BKP11R    8000  PWM1-Value at Start
RTC_BKP12R    2000  PWM2-Value at Start
RTC_BKP13R    4000  PWM3-Value at Start

RTC_BKP14R     111  Number of WS2812B LEDs

RTC_BKP17R    9600  Baud Serial2 Speed 8 D_Bytes 0 Parity 1 Stop Bits
RTC_BKP18R    9600  Baud Serial3 Speed 8 D_Bytes 0 Parity 1 Stop Bits
RTC_BKP16R   38400 Baud Serial4 Speed 9 D_Bytes 2 Parity 2 Stop Bits

RTC_BKP19R  00000000  CCS881-Baseline
Auslieferungszustand: alle Werte = 0 (d.h. nicht konfiguriert)

```

**rdump** ist wie **dump**, Speicherbereiche beziehen sich aber auf das ROM.

**else**

Bestandteil des **if – else – endif** – Blocks.

**end, end.**

**end** beendet das Programm und kehrt zum Direktmodus zurück. **end** kann auch als Stopp-Marke benutzt werden. Nach **end** kann das Programm deshalb mit **cont** fortgesetzt werden. Prozedur- und Funktionsblöcke werden ebenfalls mit **end** abgeschlossen:

```

ex. :
10000 end

```

Soll das *Hauptprogramm* beendet und auch alle **on-Interrupts** gelöscht sowie die Fortsetzung mit **cont** verhindert werden, dann kommt **end mit Punkt** zum Einsatz.

```

ex. :
10000 end.

```

**end.** bitte **ausschliesslich** am *Haupt-Programmende!* Kann aber auch im Direkt-Modus als Kommando eingegeben werden.

**endif**

Bestandteil des **if – else – endif** – Blocks.

**erase**

**erase "FileName"**

**erase** löscht Dateien auf der SD-Card. Im Kommando-Modus wird eine Warnmeldung und Sicherheitsrückfrage gestellt, die mit '**y**' (yes) beantwortet werden muss. **erase "Testprog.prg"**. Kann auch leere Verzeichnisse löschen.

exit

`exit` dient zum vorzeitigen Verlassen / Abbruch einer Prozedur oder Funktion. Kann auch statt `end` zum Beenden des Hauptprogramms eingesetzt werden. Dann aber bitte auch mit Punkt: `exit.`

fkey

`fkey Funktionstasten-Nummer, "Text"`

belegt die Funktionstasten `9 ...12` mit einem programmierbaren `Text`. Die Funktionstasten `1...8` sind vom System vorbelegt. Textlänge max. 32 Zeichen. Siehe [Anhang](#) und [`fkey\$\(\)`](#).

for

`for Var=Startwert to Endwert [step Schrittweite] : Anweisungen : next Var`

**ex. :**  
`10 for N=2 to 24 step 2` als Programm-Block  
20   print N  
30   Anweisung 2  
50   Anweisung 3  
60 next N

oder (in einer Zeile):

`80 for A=0 to pi/4 step pi/100 : print sin(A) : next A`

`for/next` ist eine Zähl-Schleife. Der Anweisungsblock zwischen `for` und `next` wird so oft wiederholt, wie durch `Startwert`, `Endwert` und `Schrittweite` bestimmt wird.

Als Zähler dient eine Variable (`Var`) `A ...Z`, die bei jedem Schleifendurchlauf um den Wert `Schrittweite` erhöht oder erniedrigt (bei negativer `Schrittweite`) wird, (mit dem `Startwert` startet) bis der `Endwert` erreicht ist. Default-Wert für `Schrittweite` (wenn `Schrittweite` nicht angegeben wird) ist: `+1`. Ist der Endwert erreicht, wird das Programm hinter der `next`-Anweisung fortgesetzt.

Mit `goto` bitte Schleifen nicht verlassen. Dafür gibt es `break`.

Mehrere (20) `for-/next`-Schleifen können ineinander verschachtelt werden. Hinter `next` darf die Zähļvariable auch weggelassen werden.

format

`format "Bezeichnung"`

formatiert die SD-Card mit FAT32 und gibt der Karte den Namen `Bezeichnung`. Der Vorgang dauert ein paar Minuten. Die Karte ist auch am PC lesbar.

## function / fU

**function Labelname** ohne Variablenliste  
**function Labelname(Variable1, Variable2,...)** mit Variablenliste

Ab Version 1.05 gibt es Programm-Label. Sprungziele können jetzt Namen haben.

<b>ex.:</b>	
10 <b>function</b> Abfrage_Eingang	das Label ohne Variablenliste
20 let result = diginp.fnvar	
30 wait 100	
<b>40 end</b>	Funktionsblock-Ende
<b>ex.:</b>	
10 <b>function</b> Abfrage_Eingang(A)	das Label mit Variablenliste
20 let result = diginp.A	
30 wait 100	
<b>40 end</b>	Funktionsblock-Ende

**function** kennzeichnet eine Programm-Sprungmarke. Es kann überall im Programm stehen, aber stets am Zeilenanfang. **function** ist ausschließlich für **User-Funktionen** reserviert. *Allgemeine* Label schreibt man mit: **proc**. **Labelname** muss mindestens 2 Zeichen lang sein und mit einem Buchstaben beginnen. Er darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen. Die Argumenteliste (Werte) des Funktionsaufrufs wird eins zu eins in die Variablenliste der Funktion übertragen. Keine weiteren Anweisungen in der **function**-Zeile. Siehe **User-Funktionen**.

get

**get Variable [,Variable , ....]** Eingabe einzelner Zeichen

holt ein (oder mehrere) Zeichen von der Konsole (Schnittstelle Nr.1 oder USB - je nach Konfiguration mit [config](#)) und speichert es entweder als Zahlenwert ([ASCII-Nummer](#)) in der [numerischen Variablen](#) oder als Zeichen in der [Stringvariablen](#). [get](#) wartet bis die Zeichen empfangen wurden. Nicht-blockierende Abfragen kann man mit [inkey](#) durchführen.

**10 get A, B(22), A\$, T(9), C\$(1)** auch möglich

get addr# get diginp# get input# get output# get relay#

Holt den Zustand von Ein-/Ausgängen eines Modbus-Moduls

```
get digin # Interface, Adresse [, Bytes], Variable holt Eingangszustände  
get input # Interface, Adresse, Eingang, Variable holt 1 Eingangszustand  
get relay # Interface, Adresse [, Bytes], Variable holt Ausgangszustände  
get output # Interface, Adresse, Ausgang, Variable holt 1 Ausgangszustand  
get adv # Interface, Adresse, Variable holt 1 Analogwert
```

**Interface** (serielle Schnittstelle): 1...4, **Adresse**: 1...32, **Bytes**: 1...247, **Variable**: jede (System)-Variable oder Array-Element, **Eingang**: 1...32, **Ausgang**: 1...32, **x**: 1...4

## 1.) get diginp # / get relay #

Kommuniziert über Interface #4, mit Modul Adresse 2 und speichert den Zustand der 8 Ein-/Ausgänge als 8Bit-Wert (1Byte) in Variable A. Wurde die Anzahl der nötigen Bytes bereits mit set mod gesetzt, kann man sich den 3. Parameter sparen – sonst bitte angeben: 16 Ein-/Ausgänge=2Bytes, 24 Ein-/Ausgänge=3Bytes usw.

2.) **get input #** / **get output #** holt den Zustand eines einzelnen Ein- oder Ausgangs, deshalb statt Byte-Anzahl die **Eingang-/Ausgangsnr** angeben. Siehe auch [Anhang](#).

3.) get **adx** #

**ex. :**

```
10 get ad3 #4, 21, $Temperatur : print $Temperatur/10
```

Holt den Messwert von Analog-Kanal 3 auf Modul 21 via Interface #4 und speichert ihn in der Variablen §Temperatur. Einige Module haben nur einen oder zwei Kanäle z.B. die Temperatur-Module. Den Messwert muss man in der Regel noch bearbeiten. Temperaturwerte z.B. durch 10 dividieren, Spannungen oder Ströme durch 100 (Integer → Fließkomma). Mehr als 4 Analogkanäle pro Modul gehen z.Z. nicht.

2.) 3.) Nicht alle Module verstehen diese Anweisungen. Die gleichnamigen System-Variablen (`adx`, `diginp`, `relay`) haben mit dieser Anweisung nichts zu tun).

get dir\$

get dir\$, Stringvariable [, Numerische Variable] [as Filter]

holt den nächsten Directory-Eintrag von der SD-Card und speichert den Namen in der **Stringvariablen**. Wird die **Numerische Variable** zusätzlich angegeben, wird dort das Datei-Attribut gespeichert. Der **Filter** bestimmt die Datei-Endung des Eintrags. pahlbasic kennt z.Z. folgende Dateitypen:

**1= .PRG            4= .TXT            7= .MP3            10= .VGM**  
**2= .PBAS          5= .BMP          8= .OGG          32= Verzeichnis**  
**3= .INC            6= .WAV          9= .MID**

**get dir\$, A\$** holt den Namen des nächsten Verzeichnis-Eintrags  
**get dir\$, A\$ as 5** holt den Namen der nächsten Bitmap-Datei in A\$

Sind alle Verzeichnis-Einträge geholt, wird ein leerer String als Eintragsname zurückgegeben. Um wieder beim ersten Eintrag zu starten, muss `reset dir$` ausgeführt werden. Das folgende Beispiel spielt nacheinander alle WAV-Dateien (max. 128 im aktuellen Verzeichnis) ab:

```
ex.:
10 proc Player
20   reset dir$ : let N=0 : clear T
30   do
40     get dir$, A$ as 6
50     if (A$ <> "") then let T(N)=A$ : inc N : ' Liste speichern *)
60   until (A$=="") or (N>126)
65
70   let N=0
75   do
80     print T(N) : play T(N) : ' Liste abspielen
85     inc N
90   until (T(N)=="")
95 end.
```

\*) Ohne Zwischenspeicherung geht es z.Z. leider nicht.

```

get #

get # Schnittstellen-Nummer, Variablen           Variable=numerisch / String
get # Filehandler, Variablen                     Filehandler=10...20 auf dem Flash
                                                Filehandler=21...24 auf der SDCard

holt (1 oder mehrere) Zeichen von der angegebenen Schnittstelle oder Datei:

get #1, A      von der seriellen Schnittstelle Nr.1 und speichert es in der Variablen A (auch vom Kombi-Interface Nr. 7: get #7, A)
get #2, A      von der MIDI-Schnittstelle, die aber auch für andere Anwendungen als MIDI genutzt werden kann.
get #3, A      von der GPS-Schnittstelle, die aber auch für andere Anwendungen als GPS genutzt werden kann.
get #4, A      von der seriellen Schnittstelle Nr.4
get #5, A      von der USB-Schnittstelle (auch: get #8, A)
get #9, A      vom USB-Keyboard (auch: get #6, A)
get #10, A     aus einer Datei mit Dateihandler 10          Dateihandler=10...24
                get# liest Bytes, die vorher mit put# in die Datei geschrieben wurden. Siehe auch fn Open().

sonst genauso wie 'normales' get. Die Schnittstellen 2, 3 und 4 werden nach dem Reset z.Z. mit 9600 Baud (konfigurierbar) gestartet.

```

## goto / go

goto Zeilen-Nummer  
goto -> Variable  
goto "Labelname"

Sprung zur Zeile  
Sprung zur Adresse  
Sprung zum Label

**ex.:**  
10 goto 100  
20 goto ->A(95)  
30 goto "Programmstart"

verzweigt zur Programmzeile 100, oder zur Adresse in A(95) oder zum Label "Programmstart", dort wird das Programm fortgesetzt.

Kann auch von der Befehlseingabe aufgerufen werden z.B. um das Programm ohne das Löschen der Variablen (wie mit run) oder ab einer bestimmten Stelle zu starten oder fortzusetzen. (nur goto Zeile und goto Label).

goto sollte maßvoll eingesetzt werden, da es den Programm-Code unübersichtlich und schwer nachvollziehbar macht. Es wurden Anweisungen hinzugefügt, die die Anwendung von goto in vielen Fällen entbehrlich machen. Siehe auch [jmp](#).

## gosub / goS

Verzweigt in ein Unterprogramm

gosub Zeilen-Nummer  
gosub -> Variable  
gosub "Labelname"

mit Zeilen-Nr.  
mit Adresse  
mit Label

**ex.:**  
10 rem Hauptprogramm  
20 gosub 100 Verzweigung zur Zeile 100  
30 Anweisung zu der zurückgekehrt wird.  
60 gosub "Unterprogramm" wiederholte Verzweigung  
70 Anweisung zu der zurückgekehrt wird.  
...  
...  
100 proc Unterprogramm Ab hier das Unterprogramm  
110 Anweisung 1  
120 Anweisung 2  
130 return Rückkehr zur Stelle, die der Aufrufenden folgt.

gosub verzweigt zur Programmzeile (hier 100) ins Unterprogramm und merkt sich die Stelle von der aus verzweigt wurde. Rückkehr mit [return](#).

Aus einem Unterprogramm können wieder Unterprogramme aufgerufen werden. Schachtelungstiefe max. 20.

goto und gosub durchsuchen den gesamten Programm-Speicher nach der Zeilen-Nummer bzw dem Label. Sie fangen entweder am Programmstart oder der goto-/ gosub-Zeile an. Daher beschleunigt eine Anordnung der Unterprogramme am Programmstart oder kurz hinter der Sprungstelle die Programmausführung. Profis verwenden Label ☺.

Eine Möglichkeit Argumente an das Unterprogramm zu übergeben, bietet die alternative Anweisung [call](#).

## gps

**gps** Einstellungshilfe  
**gps Var1, Var2, Var3, Var4** Var1, Var2 enthalten Breite, Var3, Var4 enthalten Länge  
**gps to clock [ +/- Offset ]** Uhr stellen / synchronisieren

**gps Var1, Var2, Var3, Var4** liest aus einem angeschlossenem GPS-Empfänger die Positions-Koordinaten, **gps to clock** stellt die RTC. Diese Erweiterung wurde mit UBLOX6 und -7 Modulen getestet. Das Basic wertet nur den NMEA RMC-Sentence aus, der von allen gängigen Modulen gesendet wird. Die Baudrate wird mit z.B.:

config com 3, 9600 eingestellt.

- \* Hilfe beim Positionieren der Antenne gibt die Anweisung **gps** (ohne Parameter). Es werden die GPS-Daten direkt fortlaufend auf die Konsole geleitet. Die UBLOX-Module benötigen bis zu 1 Minute, bis sie Satelliten-Daten empfangen. Diesen Vorgang mit Druck auf beliebige Taste abbrechen:

```
> gps
$GPRMC,101626.00,A,5112.24273,N,00646.39801,E,0.520,,050323,,,A*72
$GPVTG,,T,,M,0.520,N,0.963,K,A*28
$GPGGA,101626.00,5112.24273,N,00646.39801,E,1,05,4.79,147.1,M,46.5,M,,*
55
$GPGSA,A,3,19,17,12,24,06,,,,,,8.69,4.79,7.25*02
$GPGSV,3,1,12,06,18,081,20,10,04,263,,11,06,118,13,12,80,253,20*7F
$GPGSV,3,2,12,15,02,176,,17,09,036,14,19,33,052,33,24,60,133,34*7B
$GPGSV,3,3,12,25,46,253,,28,03,300,,29,04,196,,32,37,301,16*70
$GPGLL,5112.24273,N,00646.39801,E,101626.00,A,A*6B
ready
```

Auch per Software kann man mit **sys(102)** prüfen, ob überhaupt ein GPS-Signal vorliegt. Wurde der RMC-Sentence erkannt, dann ist das Ergebnis **true**, sonst ist es **0**. Der Datenstrom wird ca. 2 Sekunden überwacht.

Zum weiteren Betrieb muss das GPS-Parsing eingeschaltet werden. Dies geschieht dauerhaft mit:

config gps = on      oder nur temporär:      set gps = on

Nun kann, wer will, die Hochlaufphase des Empfängers mit Abfrage von **sys(28)** überwachen. Liefert **sys(28)** den Wert **true** zurück, sind die Daten valide. **sys(28)** prüft auf den Buchstaben 'A' im RMC-Sentence. (rot markiert)

```
> print sys(28)
4294967295
ready
```

- \*

**Var1 ...Var4** sind alle numerische Variablen oder Array-Elemente:

```
ex.:
10 gps A, B, C, D
20 print A; "'"; B; "'N,   "; C; "'"; D; "'O"
```

```
> run
51° 12.23962'N,   6° 46.34437'O
ready
```

In **A** und **B** steht die nördliche/südliche Breite: **A** (Grad), **B** (Minuten). In **C** und **D** steht die östliche/westliche Länge: **C** (Grad), **D** (Minuten). Die folgenden

Zeilen rechnen in Grad, Minuten und Sekunden um:

```
ex. :
10 gps A, B, C, D
20 let A$=str$(A)+"° "+str$(int(B))+"' "+str$((B-int(B))*60)+"'' "
30 let A$=A$+str$(C)+"° "+str$(int(D))+"' "+str$((D-int(D))*60)+"'' "
40 print A$
```

```
> run
51° 12' 14.38619'' 6° 46' 20.61059''      damit kann auch Google Maps was
                                              anfangen
```

- \* **gps to clock +/- Offset** stellt die interne RTC (Uhr) mit der Satelitenzeit:

```
> gps to clock +1      stellt die Uhr (Zeit und Datum) auf mitteleuropäische
                           Winterzeit (+2 für MEZ Sommerzeit)
Clock synchronised
ready
```

Überschreitet Uhrzeit + Offset die 0:00-Uhr-Grenze, wird eine Fehlermeldung ausgegeben. Diese kleine Utility berücksichtigt evtl. Datumswchsel während des Stellens nicht. Das Stellen der Uhr mit GPS deshalb nicht zwischen 0:00 und 3:00 Uhr vornehmen. **Offset** ist der Stundenabstand zu UTC (früher GMT). User in der Zeitzone MEZ müssen keinen Offset angeben.

Die U-Blox7-Module (mit USB-Anschluss) können recht komfortabel mit dem frei verfügbaren Tool U-Center (bis Windows10) eingestellt bzw. umkonfiguriert werden. Z.B. Stationsmodus, wenn die Module nicht bewegt werden.

## i2c

Für I<sup>2</sup>C-Bus(SensorBus)-Komponenten, für die keine systemeigenen Anweisungen oder Funktionen vorhanden sind, kann man eigene „Treiber“ schreiben.

i2c "r", Chip-Adresse, Register-Startadresse, Anzahl	Lesen
i2c "w", Chip-Adresse, Register-Startadresse, Anzahl	Schreiben

**r** – Lesemodus, **w** – Schreibmodus. Viele I<sup>2</sup>C-Sensor-Chips arbeiten nach dem Mailbox-Prinzip: Es gibt Register in die man mehrere Werte nacheinander schreiben kann oder aus ihnen lesen. Andere Chips inkrementieren stattdessen die Registeradressen automatisch.

Die Werte / Variablen sind Bytes. Wie viele Bytes und welche Werte bitte dem Datenblatt des Sensors entnehmen. Falsche Anzahl, falsche Werte werden oft mit Error quittiert. Der I<sup>2</sup>C-Bus Datenverkehr kann dann gestört sein. Wie gut, dass es den Reset-Taster gibt.

Die zu sendenden Bytes ins Array **I()** ab Index **0** schreiben. Die empfangenen Bytes stehen dann wieder im Array **I()** ab Index **0**.

Für Chips mit exotischen Protokollen ist dieses vereinfachte Verfahren nicht geeignet.

setzen von **4** Bytes in Mailbox-Register **10** (Thresholds) von CCS811-Sensor Chip-Adresse 0x5A:

```
ex. :
10 array I(0), 22, 33, 44, 55
20 i2c "w", 0x5A, 0x10, 4      4 Bytes in Array I()
                                              auf den Bus schreiben
```

Benutzung des HTU31-Temperatur-/rel.Luftfeuchtigkeits-Sensors nur mit

### Basic-Bordmitteln:

```
ex. :
10 i2c "w", 0x40, 94, 0
15 wait 100
Start einer Messung von
Temperatur und rel.
Luftfeuchtigkeit, Adresse: 0x40
kurz warten

20 i2c "r", 0x40, 0, 6
30 let A = (I(0)<<8)+I(1)
lesen von 6 Bytes von Adresse
0x40, ab Register 0
Bytes zu 16-Bit zusammensetzen

40 let B = (I(3)<<8)+I(4)
Bytes zu 16-Bit zusammensetzen

50 let T = -40+(165*(A/65535))
Temperatur berechnen
(Herstellerangabe)

60 let R = 100*(B/65535)
Luftfeuchte berechnen
(Herstellerangabe)

70 print fix(T), fix(R)
Ausdrucken mit 2 bzw.3
Kommastellen
```

Der I<sup>2</sup>C-Bus ist Timeout überwacht, deshalb werden u.U. keine Fehler angezeigt – auch wenn die Übertragung nicht geklappt hat. Fehler lassen sich nachträglich aber mit `sys(9)` abfragen.

Während der Übertragung werden die `on`-Interrupts abgeschaltet.

### i2c2

Da der Sensorbus recht langsam arbeitet, wurde ein zweiter I<sup>2</sup>C-Bus für allgemeine Anwendungen mit Standartgeschwindigkeit \*) hinzugefügt. Der I2C2-Bus muss allerdings erst aktiviert werden. Er steht nicht zur Verfügung, wenn die serielle Schnittstelle 4 benötigt wird.

Sensoren, die vom System nicht besonders unterstützt werden, können natürlich auch hier angeschlossen werden.

<code>i2c2 "r"</code> , Chip-Adresse, Register-Startadresse, <b>Anzahl</b>	<b>Lesen</b>
<code>i2c2 "w"</code> , Chip-Adresse, Register-Startadresse, <b>Anzahl</b>	<b>Schreiben</b>

**r** – Lesemodus, **w** – Schreibmodus. **Anzahl** = Anzahl Bytes, die gesendet werden sollen.

Die zu sendenden Bytes ins Array `I()` ab Index **0** schreiben. Die empfangenen Bytes stehen dann wieder im Array `I()` ab Index **0**. Es gilt das unter i2c Gesagte.

Für Chips mit exotischen Protokollen ist dieses vereinfachte Verfahren nicht geeignet.

Der I2C2-Bus ist Timeout überwacht, deshalb werden u.U. keine Fehler angezeigt – auch wenn die Übertragung nicht geklappt hat. Fehler lassen sich nachträglich aber mit `sys(9)` abfragen.

Während der Übertragung werden die `on`-Interrupts abgeschaltet.

if

#### if-Block:

```
if Ausdruck  
    [Anweisungen]  
[else]  
    [Anweisungen]  
endif
```

Die Anweisung für Entscheidungen im Programm: **if** rechnet den **Ausdruck** aus und führt die folgenden Anweisungen aus, wenn der **Ausdruck** wahr ( $\rightarrow 0$ ) ist. Ist die **else**-Anweisung vorhanden, so werden die darauffolgenden Anweisungen bei *nicht* Erfüllung ausgeführt. Bei Zeichenketten sind die Zeichenkettenvergleiche und "in" möglich. Abgeschlossen wird der Block mit **endif**.

**if** kann auch verschachtelt werden. **if**, **else** und **endif** müssen am Zeilenanfang (Einrückungen zählen nicht) stehen:

```
ex.:  
10 if milli<3300  
20     print "bitte warten"  
30 else  
40     if milli<7000  
50         let relay.2 = 1 : print "bitte passieren"  
60         wait 5000 : let relay.2 = 0  
70 else  
80     print "zu spät, passieren nicht mehr möglich"  
90 endif  
95 endif
```

#### if-Zeile:

```
if Ausdruck goto Zeilen-Nummer | Zeilen-Adresse | Zeilen-Label  
if Ausdruck gosub Zeilen-Nummer | Zeilen-Adresse | Zeilen-Label  
if Ausdruck then Anweisung1 : Anweisung2 ...
```

Aus Kompatibilitätsgründen gibt es die **if**-Anweisung mit **goto** / **gosub** und **Zeilennummer** / **Adresse** / **Label** (wie in vielen Vintage-Basic-Dialekt) ohne Programmzeilenblock. Hier **if-Zeile** genannt. **if** prüft den **Ausdruck** und verzweigt ggf. zum angegebenen Ziel:

```
ex.:  
10 if time>10 goto 200          kein Block, kein else möglich, kein endif nötig.  
                                (if goto ist 1 Anweisung)  
  
10 if time>10 goto >A          siehe auch Operator @  
  
10 if time>10 gosub "Temperatur_messen"      siehe auch Label
```

Hier prüft **if** den Ausdruck und führt – ist der Ausdruck wahr – die in der gleichen Zeile auf '**then**' folgenden Anweisungen aus – ist der Ausdruck unwahr, wird zur nächsten Zeile gesprungen:

```
10 if time>10 then print time          nun möglich  
10 if time>10 then gosub 100        (if then goto sind 2 Anweisungen)
```

## impuls / iM

`impuls relayx, time [, ImpulsTimer ]`

Impuls auf Ausgang

`x = 1 ...16,    time in ms,    ImpulsTimer = 1, 2, 3 oder 4`

**ex.:**

```
20 impuls relay3, 2000
```

Schaltet das Relais 3 für z.B. 2 Sekunden ein und danach wieder aus. War das Relais vorher eingeschaltet, wird es für die Impulszeit `time` ausgeschaltet (getoggelt). Default-Wert für `time` ist 100ms. Programm und Direktmodus. Im obigen Beispiel benutzt `impuls` keinen `ImpulsTimer` (sondern den Millisekidentimer `milli`). Das Programm hält für die Impulszeit an. `impuls relay ohne ImpulsTimer` funktioniert daher nicht in `on`-Interrupt-Routinen.

`time` max. 2147483647 ms, Auflösung und Genauigkeit ca. 0,5ms.

**ex.:**

```
20 impuls relay3, 2000, 1 : impuls relay9, 300.5, sys(65)
```

In diesem 2. Beispiel wartet das Programm nicht auf das Ende des Impulses. Funktioniert also auch in `on`-Interrupt-Routinen. Es sind 4 ImpulsTimer vorhanden, es können also 4 Impulse parallel angestoßen werden.

Wird ein neuer Impuls (auf demselben Impulskanal) gestartet bevor der vorherige Impuls beendet wurde, passiert im Run-Modus nichts bzw. im Kommando-Modus wird eine Fehlermeldung ausgegeben. Ob ein Impulskanal noch aktiv ist, kann man mit `sys(61) ...sys(64)` abfragen. `sys(65)` liefert den nächsten freien Impulskanal oder 0 wenn kein Kanal mehr frei ist.

Die Impulstimer lassen sich auch vorzeitig nach-starten (Treppenhauslicht).

`time` max. 1073741823 ms, Auflösung und Genauigkeit ca. 250µs.

Die Impulstimer können auch Interrupts auslösen. Siehe [on impuls](#).

Bitte beachten: die Relais-Nr. ist um 1 höher als die Bit-Nr. der Systemvariablen `relay`.

## inc, incr

`inc Variable [, Wert]`

```
> inc F
```

`inc` erhöht den Wert der `Variable` (hier `F`) um 1. `inc F` ist etwa 50% schneller als: `let F = F+1`. `Variable` kann auch ein Array-Element (`A()...J()`, `L()`, `P()`, `S()`) sein. Gibt man den `Wert` zusätzlich an, wird um diesen Wert erhöht. `Wert` ist eine Ganzzahl > 0 und < 32768. Das Ergebnis wird nicht geprüft.

`incr` (System-) `Variable 1`, (System-) `Variable 2`, ...

```
> incr N, M, font, Z(9)
```

`incr` kann *zusätzlich* auch Systemvariablen sowie das Array `Z()` und auch gleich mehrere Elemente auf einmal. (`incr` ist ca. 10% langsamer als `inc`). Es wird stets um 1 erhöht.

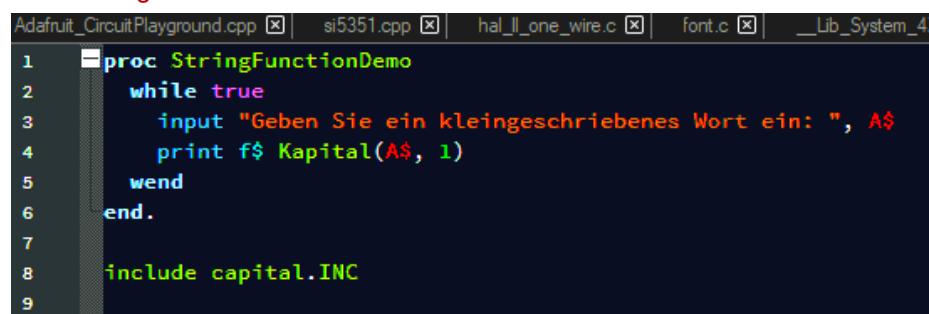
include

include Dateiname.INC

include ist keine Basic-Anweisung! Es ist eine Pseudo-Anweisung für die Instruktion **load**. Stößt **load** während des Ladevorgangs *im Basic-Text* auf die Anweisung **include**, so lädt es die angegebene Datei hinzu. **include** funktioniert nur in Basic-Textdateien (.PBAS) auf der SD-Card. Es entspricht der Präprozessor-Anweisung #include aus anderen Programmiersprachen.

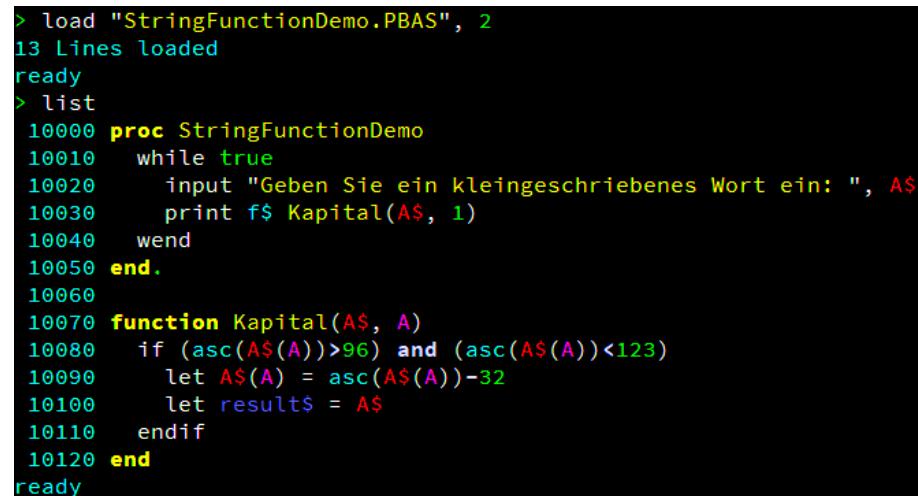
Die eingefügten Dateitexte stehen dann an Stelle von **include** im Programmspeicher. Die Include-Dateien dürfen weitere **include** Anweisungen enthalten. Der knappe RAM-Speicher setzt aber Grenzen!

Zwischen **include** und dem **Dateinamen** muss genau ein Space-Zeichen stehen. Der Dateiname darf nur aus Buchstaben, Ziffern, dem Unterstrich und dem Punkt bestehen. **include** muss direkt am Zeilenanfang im Text stehen. Die Datei-Endung muss **.INC** sein. Hier ein Beispiel mit Notepad++, Datei **StringFunctionDemo.PBAS**:



```
Adafruit_Circuit Playground.cpp | si5351.cpp | hal_ll_one_wire.c | font.c | __Lib_System_4
1  proc StringFunctionDemo
2      while true
3          input "Geben Sie ein kleingeschriebenes Wort ein: ", A$
4          print f$ Kapital(A$, 1)
5      wend
6  end.
7
8  include capital.INC
9
```

das Listing in TeraTerm – nach dem Laden mit load - sieht dann so aus:



```
> load "StringFunctionDemo.PBAS", 2
13 Lines loaded
ready
> list
10000 proc StringFunctionDemo
10010     while true
10020         input "Geben Sie ein kleingeschriebenes Wort ein: ", A$
10030         print f$ Kapital(A$, 1)
10040     wend
10050 end.
10060
10070 function Kapital(A$, A)
10080     if (asc(A$(A))>96) and (asc(A$(A))<123)
10090         let A$(A) = asc(A$(A))-32
10100         let result$ = A$
10110     endif
10120 end
ready
```

Dank **include** ist es möglich Programme zusammenzusetzen und Bibliotheken anzulegen.

input / inP

```
input ["Aufforderungstext", ] NumVar  
input ["Aufforderungstext", ] StringVar
```

Die Anweisung für Eingaben vom Terminal. **input** liest alle Zeichen ein bis die Return-Taste gedrückt wird (**input** verwendet den Zeileneditor). Dann wird der eingetippte Text an die **String**- oder **numerische** Variable übergeben. Ist die Variable numerisch, sind nur die Zifferntasten der Dezimalpunkt und das Minus-Zeichen aktiv. **input** funktioniert nicht in Interrupt-Routinen.

Eingabe eines Zahlenwerts vom Terminal, muss mit der Return-Taste abgeschlossen werden. Ergebnis in der Variablen **A**:

```
ex.:  
10 input "Geben Sie Ihr Alter ein: ", A
```

Eingabe einer Zeichenkette vom Terminal, muss mit der Return-Taste beendet werden:

```
ex.:  
10 input "Name: ", B$
```

Man kann neuerdings auch die Eingabelänge vorgeben. Dazu muss der Aufforderungstext eine "Formatierung" mit Rauten enthalten. Jede Raute steht für ein Eingabezeichen, z.B.: 4-stellig:

```
ex.:  
10 input "Ihre PIN: ####", A
```

Die maximale Eingabelänge sollte aber nicht überschritten werden:

Für dez. Zahlen:	15
für binäre Zahlen:	32
für Hex-Zahlen:	8
für Text	127

Sieht auf dem Terminal dann so aus:



input #

liest von den seriellen Schnittstellen 1...4, vom USB, vom Keyboard, aus Dateien.

**input # Nr, Var [, Anzahl] [, Delimiter]**

**Nr** = Schnittstellen- / Filehandler-Nummer siehe unter [get #](#))

**Var** = **numerische** oder **String**-Variable

**Anzahl** = Anzahl Zeichen, die empfangen werden sollen

Default = 255

**Delimiter** = ASCII-Nummer des Zeilen-Ende-Zeichens

Default = 13

#### 1.) Schnittstelle 1 ...9

**input # 1 ...9** liest die **Anzahl** Zeichen von der gewählten Schnittstelle bis zum **Delimiter**-Zeichen ein – je nachdem, was zuerst eintritt:

10 <code>input #3, B\$</code>	liest bis zu <b>255</b> Zeichen von Schnittstelle <b>3</b> ein, bis zum Zeichen <b>13</b> (Return) und speichert sie in der Variablen <b>B\$</b> .
10 <code>input #1, A\$, 9</code>	liest bis zu <b>9</b> Zeichen von der Schnittstelle <b>1</b> ein, bis zum Zeichen <b>13</b> (Return) und speichert sie in der Variablen <b>A\$</b>
10 <code>input #2, D, 15, "A"</code>	liest bis zu <b>15</b> Zeichen von der Schnittstelle <b>2</b> ein bis das Zeichen "A" auftritt, bildet eine Zahl aus den Zeichen und speichert sie in der Variablen <b>D</b> . Wird keine Zahl erkannt, wird <b>0</b> übergeben.

## 2.) Filehandler 10...20

`input #10...20` liest 'komplette' Zeichenketten aus Dateien auf dem Flash-Chip. Die Angaben **Anzahl** und **Delimiter** werden ignoriert. Beschreibung siehe [fn Open\(\)](#).

## 3.) Filehandler 21...24

wie 2.) nur auf der SD-Karte.

`input #` hat keinen Zeilen-Editor und auch kein lokales Echo. Es dient hauptsächlich der Kommunikation mit externen Geräten und Dateien, die ihrerseits ganze Zeilen senden. `input #` funktioniert nicht in Interrupt-Routinen.

### input bin

#### `input bin` Variable

Eingabe eines numerischen Werts als (max. 32-stellige) Binärzahl. `input bin` funktioniert nicht in Interruptroutinen.

```
ex.:
10 input bin A
```

### input hex

#### `input hex` Variable

Eingabe eines numerischen Werts als (max. 8-stellige) Hexadezimalzahl. `input hex` funktioniert nicht in Interruptroutinen.

```
ex.:
10 input hex "Hex-Zahl -> " , A
```

### inpt

Rechnendes `input`. Der Eingabewert muss zunächst berechnet werden? `inpt` stellt die in pahlbasic eingebaute Rechenroutine zur Laufzeit des Programms zur Verfügung:

1. `inpt ["Aufforderungstext",] NumVar`
2. `inpt ["Aufforderungstext",] StringVar`

1) *Eingabe* eines berechenbaren Ausdrucks und das Rechenergebnis einer (System-)Variablen zuweisen. Der Ausdruck wird auf Syntax geprüft!

```
ex.:
10 inpt "berechne: ", A : print " = "; : print A
```

Als Eingabe sind Ausdrücke wie `4*sin(pi/8)+.3` möglich. Diese Anweisung verwandelt den Basic-Computer in einen Taschenrechner.

- 2) *Eingabe* eines berechenbaren Ausdrucks und speichern des Ausdrucks in einer **Stringvariablen**. Der Ausdruck wird nicht geprüft! Das übernimmt `calc()`.

```
ex.:
10 inpt "berechne : ", A$ : print " = "; : print calc(A$)
```

Der String wird in Tokens umgewandelt und ändert sich daher sofort nach der Eingabe. Dies kann mit: `print A$` überprüft werden. Siehe auch [calc\(\)](#). `inpt` funktioniert nicht in Interroutinen.

## insert / inS

```
insert String1, pos, String2
```

Die Anweisung `insert` fügt `String2` in `String1` an Position `pos` ein. `String1` ist eine **Stringvariable**, `String2` jeder in pahlbasic erlaubte (auch verkettete) String oder Teilstring. Ist `pos` negativ, wird vom rechten Stringende gezählt.

```
ex.:
10 let A$ = "pahlbasic super!"
20 insert A$, 10, " ist" & " so"
30 print A$
```

```
> run
pahlbasic ist so super!
ready
```

## jmp

```
jmp Labelname
```

jmp=jump (Sprung)

wie [goto](#), als Sprungziel ist aber nur ein [proc](#) Label zulässig.

```
ex.:
10 jmp Main
```

Programmstart

```
...
...
1000 proc Main
```

Hauptprogramm

```
...
5000 end.
```

Ende

Manch einer ordnet Funktionen und Prozeduren ja lieber vor dem Hauptprogramm an – ich gehöre dazu. Dann sieht ein `jmp` zur Hauptprogrammschleife (über die Funktionen und Prozeduren hinweg) doch besser aus als ein `goto` (es sind auch keine Anführungszeichen nötig). Auch als Kommando.

## keypad / key

```
keypad NumVar oder Systemvar
keypad StringVar
```

Die Anweisung für Eingaben vom Touch-LCD. Es erscheint ein alphanumerisches Tastenfeld auf dem TFT-LCD, auf dem mit einem Stift oder Finger Eingaben gemacht werden können. Einen Fragetext wie z.B.: “Bitte Frequenz eingeben” kann man vorher in `H$` speichern. Langes Tippen

schaltet auf Großschrift bzw. Satzzeichen um. Tippen auf das = Zeichen beendet das Keypad mit der Eingabe in der Variablen. Tippt man das rote Feld an, wird das Keypad ohne Ergebnis verlassen.

Das Fragezeichen schaltet einen Taschenrechnermodus ein (ähnlich wie das ? im Direktmodus). Tippen auf das = Zeichen liefert das Ergebnis, nochmaliges Tippen auf = verlässt das Keypad mit dem Ergebnis in der Variablen. Siehe [Anhang](#).

**ex.:**  
10 let H\$ = " geben Sie den Code ein "  
20 keypad A Eingabe einer num. Variablen

**ex.:**  
10 let H\$ = " geben Sie den Code ein "  
30 keypad A\$ Eingabe einer Zeichenkette

Die 'Tasten' sind nicht animiert, man sieht aber trotzdem eine Reaktion auf jedes Antippen.

kill

```
kill "Datei-Name"  
kill Datei-Nummer
```

löscht die angegebene Datei auf dem Flash-Chip. `kill` funktioniert wie `del`. Kleiner Unterschied: `kill` löscht ausschließlich sequentielle Daten-Dateien. Innerhalb von Programmen erfolgt keine Rückfrage.

lcd

Wahl des Ausgabemediums für `print`

Lcd Schalter Schalter = 0, 1, 2 oder 3

**ex.:**

10 lcd 0 : print "Hallo Welt"	Ausgabe auf Terminal
20 lcd 1 : print "Hallo Welt"	Ausgabe auf TFT-LCD

**Icd 1** leitet `print`-Ausgaben vom Terminal auf das LCD um. **Icd 0** schaltet wieder zurück zum Terminal. `print` nutzt auf dem LCD ausschließlich `font 2` (Courier New 8x16 – sehr klein), die Textfarbe mit `set print color` oder `color$()` einstellen. `cls$` und `loc$()` funktionieren auch. Siehe auch Anweisungen: `text`, `print#` 9.

lcd 2

schaltet die Display-Beleuchtung aus. `Lcd 3` schaltet sie wieder ein. Die Hardware-Voraussetzung: Der entsprechende Port-Pin (PE12 - `TFT_BLED`) muss mit dem Display (Pin `BL` oder `LED`) verdrahtet sein.

## led

Steuert einen LED-Streifen-/Panel mit max. 256 adressierbaren WS2812B-LEDs an. **Achtung!** Bei voller Helligkeit nicht direkt in die LEDs schauen!

led LED-Nummer, Rot, Grün, Blau [, Aktuell]	LED ein-/ausschalten
led << Anzahl	LEDs nach links schieben
led >> Anzahl	LEDs nach rechts schieben
led !	LED-Farben aktualisieren
led String, Rot, Grün, Blau	Text aufs Panel, in Entwicklung
led 0 oder led off	alle LEDs aus
led on, Rot, Grün, Blau	alle LEDs mit Farbe einschalten <b>Achtung! Strom beachten!</b> 256 Stück WS2812B-2020-LEDs benötigen > 4A bei Rot, Grün und Blau jeweils 255

LED-Nummer: 1 ...256  
Rot-Anteil: 0 ...255  
Grün-Anteil: 0 ...255  
Blau-Anteil: 0 ...255  
Aktuell: 0 oder 1

Auf einem LED-Streifen mit max. 256 LEDs (WS2812B) können alle LEDs einzeln mit ihren Farbwerten eingeschaltet werden. Jeder LED kann man die 3 Farbwerte zuweisen.

Verschoben werden alle 256 LEDs um **Anzahl** LEDs nach links oder rechts.

led als Kommando:

led 1, 40, 0, 50	LED Nr.1 Violett einschalten
------------------	------------------------------

led im Programm:

```
ex. :  
10 led 5, 20, 0, 30 : led 4, 0, 0, 170 : led 3, 0, 30, 0  
20 led 2, 50, 10, 0 : led 1, 50, 0, 0  
30 do : led << 1 : wait 50 : until
```

Dieses Beispiel schaltet die LEDs 1...5 mit verschiedenen Farben ein und schiebt diese „5er-Kette“ nach links über den LED-Streifen und rechts wieder herein. Sind **Rot = 0**, **Grün = 0** und **Blau = 0** ist die LED aus. Weiße LEDs erhält man mit **255** für **Rot**, **Grün** und **Blau**.

Die LEDs sind sequentiell hinter einander geschaltet, die Übertragung der Farbdaten der 256 LEDs dauert ca. 10ms. Für manche Anwendung ist das evtl. zu langsam, daher kann man die sofortige Aktualisierung der Farbwerte mit Hilfe des Zusatzparameters **Aktuell** abschalten: **led 1, 40, 0, 50, 0** speichert nur den Farbwert für LED 1 für eine spätere Übertragung. Aktualisieren kann man die Farbwerte dann mit einer **led**-Anweisung bei der **Aktuell = 1** ist oder ohne den **Aktuell**-Parameter oder aber mit der Anweisung **led !**:

```
ex. :  
5 do  
10   for N=1 to 256  
20     led N, 63 and rand, 63 and rand, 63 and rand, 0  
30   next N  
40   wait 1000  
45   led !  
50 until
```

Erzeugt ein jede Sekunde wechselndes Zufallsmuster auf dem LED-Streifen.

Seit V105A26 kann man auf die Farbwerte der LEDs auch über das Array **L()**

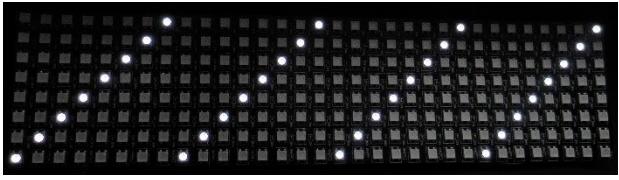
zugreifen. Das Array `L()` ist 256 mal 32-Bit groß. Der Farbwert von LED 1 steht in Array-Element `L(0)` und der von LED 256 in `L(255)`.

Dieses neue Array erleichtert auch die Darstellung von Zeichen und Mustern auf LED-Panels (mit 32 Spalten à 8 LEDs) dann aber mit 2 Dimensionen (für Spalte und Zeile). Der Farbwert von LED 1 steht dann in Element `L(0, 0)` und der von LED 256 in `L(31, 7)`. Siehe auch [Arrays](#).

Man schreibt zunächst die Farbwerte in die Array-Elemente und sendet sie danach mit `led !` an das Panel.

```
ex. :
5  for M=0 to 3
10   for N=0 to 7
20     let L(N+(M*8), N) = fn Rgb2(4,4,4)      zeichnet Linien auf das Panel
30   next
40 led !
50 next
```

Während der Übermittlung der Farbdaten werden alle Interrupts abgeschaltet. (Das Timing ist sehr sensibel). Die betroffenen Timer-Funktionen sollten im Zusammenhang mit der Anweisung `led` vermieden werden, bzw. deren größere Ungenauigkeit in Kauf genommen.



Es gibt auch LED-Streifen, deren LEDs eine abweichende Farbreihenfolge haben, evtl. ausprobieren.

Für die korrekte Funktion sollte die genaue Zahl der angeschlossenen LEDs mit [config led](#) eingestellt werden.

let

```
let NumVar = (Rechen-)Ausdruck
let Stringvar = (Teil-)Zeichenkette
```

```
ex. :
10 let A = 2*(N+sin(pi/8))          rechnet Ausdrücke (Formeln) aus.

let N = 5>4                         Kann vergleichen

20 let C = A or (255 and B)          Logische Verknüpfungen

30 let A$ = "Hallo Welt"            Stringkonstante in eine Stringvariable

let A$ = B$                          Kopiert Strings: B$ nach A$

40 let A$ = C$(5, 4)                 Kopiert 4 Zeichen von C$ ab dem 5.
Zeichen nach A$

50 let D$ = clock                   Kopiert Uhrzeit nach D$

let relay.1 = 1                      Schaltet Relais 2 ein

60 let D$(3) = "A"                  Tauscht ein einzelnes Zeichen aus.

let G$ = date _ clock + " Uhr"    verkettet Strings
```

Weist einer **String-** oder **numerischen** Variablen einen Wert zu. Kann auch direkt (im Kommando-Modus) eingegeben werden.

**Negative Werte bitte in Klammern:** `let A = (-2.33) + (-1.4)`

Ausdrücke werden strikt von links nach rechts berechnet, Punkt-vor-Strich-Rechnung ist nicht implementiert. Bitte Klammern setzen zur Änderung der Reihenfolge: `let N = 3+(4*A)`. Max. 20 Klammerebenen.  
Siehe auch [Stringfunktionen](#).

### `list / ll, rlist`

\* `list` Zeile - Zeile

**ex. :**

<code>list</code>	listet das gesamte Programm
<code>list 10</code>	nur Zeile 10
<code>list -100</code>	bis Zeile 100
<code>list 100-</code>	ab Zeile 100
<code>list 20-100</code>	die Zeilen 20 bis 100

Zeigt die gewünschten Programmzeilen an. Dabei wird eine dieser Beschreibung ähnliche Syntax-Einfärbung verwendet. `list` alleine listet das ganze Programm. `list 10` nur Zeile 10. `list 20-100` die Zeilen 20 bis 100, `list 20-` alles ab Zeile 20. Sollte nicht im Programm verwendet werden.

Die letzte gelistete Zeile kann man – zur Bearbeitung – mit **Strg + p** (**ctrl p**) wieder in den Eingabepuffer holen und sofort editieren.

### Ausgabe auf dem TFT-LCD:

Werden mehrere Programm-Zeilen angezeigt, ist der Zeilenumbruch abgeschaltet. Der Zeileninhalt wird möglicherweise nicht vollständig dargestellt. Einzelne Zeilen (`list 100`) werden immer komplett angezeigt, auch über bis zu 3 Display-Zeilen und können auch über diese (3) Display-Zeilen editiert werden (auf dem 800x600-Display: 1,5 Display-Zeilen entspr. 123 Zeichen).

\* `rlist` ist exakt wie `list` jedoch für den internen [ROM-Speicher](#). Das gelistete Programm wurde zuvor z.B. mit `save@` oder `rload` gespeichert.

## load / lO, rload

load Name [, Medium]	Medium: 1=Flash, 2=SD-Card
load Programm-Nummer	nur Flash Chip, 0-31 (bei 2MByte)
load @	
load !	
rload Name [,1]	nur SD-Card, [1=Autostart]

load lädt die Programm-Datei mit dem angegebenen **Namen** oder ihrer **Nummer** (mit dir ermitteln) in den RAM-Speicher. Von einem Programm aus, kann man Programme nachladen, die werden sofort gestartet, ohne die Variablenwerte zu löschen (das ursprüngliche Programm im Arbeitsspeicher wird überschrieben). Siehe auch save, del und dir.

load [0] lädt entweder die versteckte Programmdatei 0 oder – wenn kein serieller Flash-Speicher vorhanden ist – die im ROM befindliche Programmdatei. Letzteres lässt sich mit config load auch bei vorhandenem seriellen Flash erzwingen. Die SD-Card kennt *keine* Programm-Nummern.

> load "Temperaturlogger"	lädt vom Flash-Chip
> load 17	lädt vom Flash-Chip
> load "Templogger.prg", 2	lädt von der SD-Card

load @ lädt die im internen ROM befindliche Programmdatei ins RAM unabhängig von Einstellungen mit config. Nur für Chips mit 1Mbyte ROM.

load ! lädt nur die mit save @ oder save ! ins interne ROM gespeicherten Einstellungen ins config-Backup-Memory, evtl. ist ein Reset nötig. Nur für Chips mit 1Mbyte ROM.

rload Name lädt die angegebene Datei von der SD-Card in das ROM des µC. Nur .PBAS-Dateien, nur im Kommando-Modus. include funktioniert auch hier. Konfigurationen werden nicht gespeichert. Nur für Chips mit 1Mbyte ROM.

## locate / loC

locate Zeile, Spalte

```
ex.:
10 cls : locate 1, 5 : print "pahlbasic V1.05"
20 wait 1500
30 cls : locate 10, 30 : input "Ihre PIN: #####", A
40 if A=1234 then locate 12, 30 : print "Willkommen Hr. Pahl"
```

Setzt den Cursor auf dem Terminal oder TFT-LCD auf Zeile und Spalte abhängig von einer evtl. Umleitung mit lcd. Zeile = 0 ... 19, Spalte = 0 ... 52 auf dem TFT \*) - auf dem Terminal gilt: Zeile = 1 ... 24 und Spalte = 1 ... 123

\*) Auf LCDs mit 320x240 Pixel: Zeile 0 ... 14, Spalte 0 ... 34.

## max

max Array-Element1 to Array-Element2, Ergebnis-Variable

```
ex.:
10 for N=0 to 9 : let A(N) = rand and 255 : next
20 'Array mit Zufallszahlen füllen
30 max A(0) to A(9), X                         Maximum ermitteln
40 print X                                       Ergebnis drucken
```

**max** ermittelt das Maximum in einem **Array-Abschnitt** und speichert es in der **Ergebnis-Variablen**. Der Array-Abschnitt darf von **A(0)** bis **J(99)** reichen, das **2. Array-Element** muss aber *hinter* dem **1. Array-Element** liegen.

md

(make Directory)

**md "BasicDateien"**

legt das Verzeichnis **BasicDateien** auf der SD-Karte an. Die Verzeichnis - struktur sollte nicht allzu komplex werden. Der Basic-Computer ist kein PC.

mean

berechnet das arithmetische Mittel über einen Array-Abschnitt

**mean Array-Element1 to Array-Element2, Ergebnis-Variable [, Spezial]**

**ex.:**

```
10 for N=0 to 19 : let A(N) = 10*(N +1) : next N  
20 'Array mit Zahlenreihe füllen  
30 mean A(0) to A(19), X  
40 print X
```

arith. Mittel ermitteln  
Ergebnis drucken: 105

**mean** berechnet das arithmetische Mittel über einen Array-Abschnitt und speichert es in der **Ergebnis-Variablen**. Der Array-Abschnitt darf von **A(0)** bis **J(99)** reichen, das **2. Array-Element** muss aber *hinter* dem **1. Array-Element** liegen.

Enthält der Array-Abschnitt z.B. Messwerte, so gibt es gelegentlich 'Ausreißer': ein oder 2 Messwerte 'tanzen' aus der Reihe, sind viel zu hoch oder zu niedrig. In diesen Fällen kann es helfen, den Schalter **Spezial** zu setzen:

Im obigen Beispiel Zeile 30 ändern:

```
30 mean A(0) to A(19), X, 3
```

arith. Mittel speziell  
ermitteln

**mean** wirft in diesem Beispiel den größten und den kleinsten Wert hinaus und berechnet das Mittel über die restlichen Werte. Es müssen mindestens 5 Werte im Array-Abschnitt stehen.

**Spezial** kann folgende Werte annehmen: **1**, **2** oder **3**:

- 1=der größte Wert wird entfernt
- 2=der kleinste Wert wird entfernt
- 3=der größte und der kleinste Wert wird entfernt

## midi

### midi Channel, Message, W1, W2

Channel=1...16 (MIDI-Kanal) W1, W2 sind Werte 0...127

Message:	8	Note Off	W1=Midi-Noten-Nr., W2=Velocity
	9	Note On	W1=Midi-Noten-Nr., W2=Velocity
	10	After Touch	W1=Midi-Noten-Nr., W2=Velocity
	11	Control Change	W1=Controller-Nr., W2=Wert
	12	Program Change	W1=Program Nr.
	13	Channel Pressure	W1=Pressure Value
	14	Pitch Wheel	W1=obere 7Bits, W2=untere 7Bits

ex.:

```
10 midi 1, 9, 44, 127 : wait 1000 : midi 1, 8, 44, 0
```

Note E4 auf Channel 1 mit vollem Anschlag starten und nach 1 Sekunde stoppen.

## min

### min Array-Element1 to Array-Element2, Ergebnis-Variable

ex.:

```
10 for N=0 to 9 : let A(N) = rand and 255 : next N
20 'Array mit Zufallszahlen füllen
30 min A(0) to A(9), X
40 print X
```

Minimum ermitteln  
Ergebnis drucken

min ermittelt das Minimum in einem Array-Abschnitt und speichert es in der Ergebnis-Variablen. Der Array-Abschnitt darf von A(0) bis J(99) reichen, das 2. Array-Element muss aber hinter dem 1. Array-Element liegen.

## morse / mO

### morse String

ex.:

```
10 morse "pahlbasic ist super"
20 let A$ = "Stationsname xyz" : morse A$
30 morse date _ clock
```

Gibt den String als (Standard) Morsezeichen am Soundausgang oder Relais 16 aus. String kann jede in pahlbasic erlaubte Zeichenkette (Literal in Anführungszeichen, Stringvariable, Teilkette oder Stringverkettung sein. (Non-ASCII-Zeichen sowie das Signal EEEEEEEE sind nicht implementiert). Die Ausgabegeschwindigkeit wird mit der Systemvariablen speed eingestellt. Siehe speed.

Ist die Soundausgabe (noch) nicht aktiv, wird der Soundausgang initialisiert, was ca. 0,5 Sekunden dauert.

Mit der Anweisung config morse kann die Ausgabe auch auf den (Relais-) Ausgang Nr.16 umgeleitet werden, um Sendestufen zu tasten.

## new

(alles neu)

```
> new
```

löscht das im RAM-Speicher befindliche Programm und alle Variablen. Nur Direktmodus. Die Arrays P(), S(), Z() werden nicht gelöscht - Farbpalette, Soundspeicher und batteriegepuffertes Backup-RAM löscht man mit clear.

next / nE

Bestandteil der **for** / **next** – Schleifenstruktur. Siehe [for](#).

on

auf Ereignisse reagieren:

on <b>adx</b> untere Schwelle, obere Schwelle start Label	adx = ad1...ad4
on <b>clockx</b> , Tag, Stunde, Minute, Sekunde start Label	x = 1 oder 2
on <b>com x</b> start Label	x = 0 ...5 oder 9
on <b>diginpn</b> start Label	n = 1 ...5
on <b>err gosub Zeilennummer   Label</b>	
on <b>impuls Timer-Nummer, Zeit</b> start Label	Zeit=1 ...1073741823 ms
on <b>millix, Zeit</b> start Label	x = 1 ...3
on <b>time Zeit</b> start Label	Zeit=1 ...4294967295s
on <b>touch</b> start Label	

**ex.:**

```
10 on time 300 start Intrpt
20 on diginp4 start Check_Inp
30 on com 0 start Uart
40 on touch start Test
```

**on** meldet eine Interrupt-Service-Routine an, die bei Auftreten eines *Ereignisses* das Hauptprogramm unterbricht und eine kurze Re-Aktion ausführt. Die *Interrupt*-Routine selbst beginnt mit **proc Labelname** und endet mit **end**.

**on** funktioniert *nur* mit **Labels**. **on** speichert die *Adressen* der **Labels** für einen schnelleren Aufruf. Daher können die *Interrupt*-Routinen ohne Geschwindigkeitsverlust auch am Programm-Ende platziert werden. Und: *Interrupt*-Routinen sollten **kurz** und **schnell** sein. Alter Programmierer Spruch: "Lungere nicht so lange im Interrupt herum!". Warteschleifen im Hauptprogramm sind niemals ein Problem, in den Interrupts aber schon. *Interrupt*-Routinen sind Prozeduren, es sollten keine weiteren Prozeduren innerhalb der *Interrupt*-Routinen aufgerufen werden.

'Langsame' Namen-Variablen funktionieren in *Interrupt*-Routinen nicht. Lokale Variablen, wie in normalen Prozeduren gibt es nicht.

Seit Version 1.05 sind **on-Hardware-Ereignisse** *vollständig* Interrupt-gesteuert. Das bedeutet: der Basic-Computer reagiert auf Ereignisse sofort, unabhängig von blockierenden Anweisungen im Hauptprogramm. (das können nicht viele µC-Basic-Interpreter von sich sagen ☺ \*)

**on**-Interrupt wieder abschalten? Das geht so:

**on Ereignis off**

Ereignis steht für: **adx, bufx, diginp<sub>x</sub>, err, impuls x, millix, time, touch**

**ex.:**

```
100 on diginp3 off
110 on milli2 off
```

\*) Alle Interrupts können das Hauptprogramm und sich auch gegenseitig jederzeit unterbrechen. Das hat nicht nur Vorteile: **print**-Ausgaben könnten zerrissen oder verstümmelt werden. Problematisch sind in der Regel auch VT100-Steuersonden, das betrifft z.B. die Anweisungen **put**, **print**, **cls** und **locate**. Daher sind Konsolenausgaben innerhalb der Interrupts deaktiviert. Die zu druckenden Strings können aber problemlos im Interrupt zusammengezettzt werden und im Hauptprogramm ausgedruckt. Einige blockierende

Anweisungen / Funktionen (`input`, `wait`) sind ebenfalls deaktiviert oder warten nicht (`get`, `inkey$`).

Werden Anweisungen oder Funktionen eingesetzt, die die `on`-Interrupts *abschalten* (`i2c`, `led`, `load`, `sensor`), müssen passende Maßnahmen getroffen werden damit keine Unterbrechungsanforderungen 'verloren gehen'.

Wird das laufende Programm beendet (mit `end` oder `ctrl + c`), dann werden die Interrupts nur inaktiviert (damit das Programm mit allen Interrupts mit `cont` fortgesetzt werden kann). Sollen die Interrupts komplett gelöscht werden, bitte das Programm mit `end.` beenden oder Anweisung `reset intrpt` ausführen.

Fehler in den Interrupt-Routinen können instabile Zustände herbeiführen. Nach dem Reset ist aber alles wieder gut. Programme in der Entwicklungsphase sicherheitshalber öfters speichern. Siehe auch Beispirogramm "interrupt\_Test".

### Die Ereignisse im Einzelnen:

`on adx`

`on adx, untere Schwelle, obere Schwelle start Label`

aktiviert ein Ereignis sobald die am angegebenen Analogeingang anliegende Spannung den Bereich (Fenster), der durch `untere Schwelle` und `obere Schwelle` festgelegt ist, verlässt. `adx` steht für `ad1` bis `ad4`. Werte für obere und untere Schwelle: `0 ... 4095` – entsprechen `0 ... 3.3V` am Eingang. Es kann nur *ein* Eingang überwacht werden:

```
ex.:
10 on ad2, 1000, 2000 start Int          Poti an ad2 anschliessen
20 do : until                            simuliert Hauptprogramm
90 end.

...
100 proc Int                           hier beginnt die Interrupt-Routine
110   if ad2<1000 then set relay1
120   'Relais 1 schaltet ein, wenn...
130   if ad2>2000 then reset relay1
140   'Relais 1 schaltet aus, wenn...
190 end
```

Für die evtl. notwendige Hysterese muss der Anwender selbst sorgen. Das Beispiel zeigt, wie's geht.

**on clock**      **on clock1, Tag1, Stunde1, Minute1, Sekunde1 start Label1**  
**on clock2, Tag2, Stunde2, Minute2, Sekunde2 start Label2**

Diese beiden Interrupts stehen nur zur Verfügung, wenn die automatische Sommer-/Winterzeitumschaltung deaktiviert wurde. Die **clock**-Interrupts werden von der Hardware-RTC (Real Time Clock) gesteuert. Gibt man den Tag mit **0** an, findet der Aufruf täglich statt. Mit Tagesangabe wird der Interrupt einmal im Monat gestartet. (Vorausgesetzt, der Monat hat die entsprechenden Tage) Soll das Ereignis einmalig sein, muss der Interrupt im Programm wieder ausgeschaltet werden:

```
on clock1 off - (beendet clock1 Interrupt)
on clock2 off - (beendet beide Interrupts) oder löschen: clear clock
```

Bitte zuerst Alarm **clock1** starten, dann **clock2**. Wird nur *ein* Alarm benötigt, Alarm **clock2** starten. (Die Alarne werden gemeinsam mit **clock2** aktiviert). Sind die Alarmzeiten gleich, wird nur *ein* Alarm (**clock2**) gestartet:

```
ex.:
5 rem clock interrupt
10 print "Einschaltzeit:"
20 input "Stunden: ", A(1)
30 input "Minuten: ", A(2)
40 input "Sekunden: ", A(3)
50 on clock1, 0, A(1), A(2), A(3) start Ein      1.Alarm
60 print "Ausschaltzeit:"
70 input "Stunden: ", A(4)
80 input "Minuten: ", A(5)
90 input "Sekunden: ", A(6)
100 on clock2, 0, A(4), A(5), A(6) start Aus    2.Alarm
200 do : until
900 end.
910 '
1000 proc Ein
1010   set relay1                                Relay1 einschalten
1100 end
1110 '
2000 proc Aus
2010   reset relay1                             Relais1 ausschalten
2100 end
```

**on com x**      **on com x start Label**

**com x** steht für **com 0** (Konsole), **com 1** (serielle Schnittstelle 1), **com 2** (MIDI), **com 3** (GPS), **com 4** (serielle Schnittstelle 4), **com 5** (USB) oder **com 9** (USB-Keyboard). **on com x** löst ein Ereignis aus, wenn mindestens ein Zeichen von der angegebenen Schnittstelle empfangen wurde. Lässt man die Schnittstellen-Nummer weg (oder gibt Nr. **0** an), wird automatisch die Schnittstelle ausgewählt, an der die Konsole hängt:

```
ex.:
10 on com 0 start Uart                         Dann hauen wir mal auf die
20 do : until                                     Tasten
90 end.
...
100 proc Uart
110   toggle relay1                            Relais 1 schaltet hin und her
190 end                                         bei jedem Tastendruck
```

**on diginp<sub>x</sub> start Label**      **x=1 ...5**

**on diginp1 ...on diginp5** wird aktiviert, wenn der entsprechende Eingang **diginp1 ...diginp5** den Ruhezustand (je nach Konfiguration mit config) verlässt:

<b>ex. :</b>	
10 <b>on diginp3 start Interrupt</b>	Taster an Eingang 3
20 config diginp=0, sys(33) or 4	Pulldowns, Totzeit ein
30 do : until	Simulation Hauptprog
40 end.	Hauptprogramm Ende
...	
1000 <b>proc Interrupt</b>	Interrupt-Routine
1010    toggle relay5	Relay 5 "Eltako"
1040 <b>end</b>	fertig

Um zu verhindern, dass ein noch anstehender Impuls den Interrupt mehrfach auslöst, ist ein Totzeit-Timer implementiert, der nach der Startflanke 10ms Prellzeit und dann 190ms lang keinen weiteren Impuls zulässt, optimiert für den Anschluss von Tastern und mechanischen Kontakten, er wird in Zeile 20 eingeschaltet.

**on diginp<sub>x</sub>, Zeichencode**      **x=1 ...5**

manchmal muss eine Steuerung ausgeführt werden, die durch einen Tastendruck auf dem Terminal oder dem Keyboard ausgelöst wird (siehe oben: **on com x**). Was aber, wenn keine Tastatur/Terminal angeschlossen ist? Kann man dafür nicht auch die digitalen Eingänge einsetzen? Doch kann man! Mit **on diginp<sub>x</sub>, Zeichencode** wird dafür ein Interrupt eingerichtet, der ein **Zeichen** in den Terminal-Eingabe-Puffer schreibt, wenn der entsprechende digitale Eingang angesteuert wird, genau so, als ob das **Zeichen** auf der Tastatur eingetippt wurde.

**Zeichencode** ist der ASCII-Code des Zeichens. Kann auch so geschrieben werden:

<b>ex. :</b>	
10 <b>on diginp3, ("p")</b>	Taster an Eingang 3, Zeichen 'p'

Es können auch Zeichencodes wie z.B. 3 (**ctrl+c**) verwendet werden:

<b>ex. :</b>	
10 <b>on diginp3, 3</b>	Taster an Eingang 3, Zeichen ctrl+c

der Totzeit-Timer wird hier automatisch aktiviert. Die gespeicherten Zeichen lassen sich auch abfragen: sys(121) ...sys(125).

**on err**      **on err gosub Zeilen-Nummer | Label**

Bei Auftreten eines Fehlers wird zur angegebenen **Programmzeile / Label** verzweigt. Dort kann der Fehler „behandelt“ werden. Der normale Programm-Abbruch im Fehlerfall wird ausgeschaltet. Ist in der Fehlerbehandlungsroutine jedoch selbst ein Fehler, wird das Programm dennoch abgebrochen. Mit **resume** wird hinter die Fehlerstelle zurückgekehrt, mit **resume erl** zur Zeile mit dem Fehler, mit **resume Zeilen-Nummer** oder **resume Label** zu einer beliebigen Programm-Zeile.

```

ex.:
10 on err gosub "Fehlerbehandlung"
20 do
30   cls : locate 10, 10
40   input "Eingabe : ",A : let B= 1/A : locate 11,10 : print B
50   'testweise 0 eingeben.
60   wait 1500
70 until
80 end.
90 '
100 proc Fehlerbehandlung
120 resume 30

```

**on impuls**

**on impuls Timer-Nummer, Zeit start Label**

Timer-Nummer: 1...4  
Zeit=1 ...1073741823 ms

verbindet Interrupt-Routinen mit den **4** Impuls-Timern. (After-Timer). Die Interrupt-Routinen werden *nach* Ablauf der Timer einmalig gestartet:

```

ex.:
10 input "Bitte PIN eingeben: ", A
20 if A=1234
30   set relay1,2 : on impuls 1, 5000 start Impi      starten
40 endif
50 goto 10                         sofort wieder zur Eingabe
90 end.
95 '
1000 proc Impi                  ahmt die Anweisung impuls relay nach
1010   reset relay1,2
1020 end

```

on milli

Es gibt jetzt 4 Timer (Every-Timer) für quasi-parallele Aufgaben:

on millix, msec start Label

x = 1 ... 4

msek = 10 ...4294967295ms

startet den Interruptx zyklisch alle msek Millisekunden. Die Interrupt-Routinen sollten in dieser Zeit auch abgearbeitet sein, sonst friert das Basic ein.

on time

## on time T-Wert start Label

löst ein Ereignis aus, wenn `time` den Wert **T-Wert** erreicht hat. Bei Aufruf von `on time` wird die Systemvariable `time` neu gestartet. `time` wird nach dem Einschalten des Basic-Computers jede Sekunde eins weiter gezählt.

Für einen zyklischen Aufruf des Unterprogramms muss die erste Anweisung des Unterprogramms `let time = 0` sein.

<b>ex.:</b>	
10 <b>on time 1 start</b> Int	Ereignis definieren
20 do : until	simuliert Hauptprogramm
30 end.	Hauptprogramm Ende
...	
...	
100 <b>proc</b> Int	Hier wird das Ereignis abgearbeitet
110 let time = 0	zyklischer Aufruf
110 toggle relay 1	Software 'Blink-Relais'
150 <b>end</b>	Prozedur-Ende

on touch

wird aktiviert, wenn das Touch-TFT-Display berührt wird:

**ex.:**

```
10 on touch start Intrpt           Interrupt starten
20 do
30   locate 10,10: print X, Y: wait 100    Koordinaten
40 until
50 end.                                ausdrucken
...
...
1000 proc Intrpt
1010   touch X, Y
1020 end                                Hauptprogramm Ende
                                         Touch-Interrupt beginnt hier
                                         Koordinaten ermitteln
                                         Prozedur-Ende
```

`on touch` hat das gleiche Timing wie `on diginp` mit Totzeit-Timer.

`play`

`play "Dateiname"`

spielt eine Sounddatei über den chip-internen DAC ab. Es können nur **.WAV**-Dateien (von SD-Card) abgespielt werden (es gilt das für Bitmaps Gesagte). Nur 16Bit PCM-Daten in Stereo oder Mono mit 11025 bis 48000Hz Samplefrequenz. Die Lautstärke kann (vorher) mit volume geändert werden. Während des Abspielens sind folgende Tasten aktiv:

<code>+/-</code> stellen die Lautstärke ein,	<code>'f'</code> springt vorwärts,	<code>'b'</code> springt rückwärts,
	<code>'2'</code> springt in die Mitte,	<code>'&lt;'</code> springt an den Anfang
<code>'u'</code> erhöht die Spielgeschwindigkeit,	<code>'d'</code> senkt die Abspielgeschwindigkeit,	
	<code>'o'</code> stellt Originalgeschwindigkeit ein,	
<code>'h'</code> hält das Spiel an,	<code>'p', 'h'</code> setzen das Spiel fort,	<code>'s'</code> stoppt das Spiel

Das Abspielen findet z.Z. im Vordergrund statt, das Programm hält also für die Spielzeit an. Nur mit STM32F405/7, beigelegte Datei **LVB.WAV** bitte auf SD-Card kopieren. Nur zusammen mit ILI9488/SSD1963. Siehe record.

`play "LVB.WAV"`

Dateiname darf jeder in pahlbasic erlaubte String sein

#### Sounderzeugung:

`play` spielt auch musikalische Noten (max. 6 parallel). `play` sucht sich die nächste freie Stimme und spielt den angegebenen Ton mit der angegebenen Länge. Für den Ablauf des Spiels sorgt der ADSR-Hüllkurvengenerator. Syntax:

`play Note1, Dauer1 [, Note2, Dauer2, ..., Note6, Dauer6]`

Note ist die MIDI-Noten-**Nummer**, Dauer in 16-tel-Noten.

z.B.:  $\frac{1}{4} = 4$  16-tel-Noten,  $\frac{3}{4} = 12$  16-tel-Noten,  $\frac{1}{2} = 8$  16-tel-Noten

```
set sound
play 70, 8, 73, 8, 76, 8, 79, 8
```

`play` kann alle 6 Stimmen parallel abspielen. Ist keine Stimme frei, bricht `play` die erste Stimme ab (und die weiteren, wenn nötig). Für die Notendauer ist der Hüllkurvengenerator zuständig (nicht `play`), deshalb müssen die Werte für tempo, attack, decay, sustain, release und volume vorher bereits eingestellt, sowie ein Sound in den Samplespeicher geladen sein (z.B. mit set sound). Bei langer **Release-Phase** ist kein Unterschied zwischen kurzen und langen Noten zu hören. (wie beim Piano, wenn das Dämpferpedal getreten wird).

#### **Wertebereiche:**

Note: 12...107, Dauer 1...8000 ( $> 7999 = \text{ewiger Ton}$ )

Dauer = 0 bewirkt, dass das Spiel der Note abgebrochen wird. (`play 70, 0`)

Für den rythmischen Aufruf der Anweisung `play` ist der User verantwortlich (z.B. mit on-milli-Interrupt). `play` (alleine) spielt keine Melodien ab. (siehe auch Beispieldatei "play-Test" und Hardwaretip: Audio-Anschluss)

Wird set sound mit Parameter aufgerufen, verhält sich `play` anders:

Während des Abspielens der Noten wird die Klangfarbe der Noten verändert durch FM-Synthese. `play` wartet in diesem Fall auf die Beendigung der Decay-Phase. In Interruptroutinen ignoriert `play` deshalb die Parameter-Angabe. Die Ausgabe von simplen Klängen z.B. Alarmsounds wird dadurch aber vereinfacht. Siehe Beispieldatei "[AlarmSound.PBAS](#)".

```
set sound 150, 5
play 70, 8, 73, 8, 76, 8, 79, 8
```

## plot

setzt ein Pixel auf dem TFT-Display. Die Anweisung `plot` ist weitgehend identisch mit `draw "p"`. Die Anweisung `plot` ist jedoch für höhere Ausführungs geschwindigkeit ausgelegt. Keins der Parameter wird überprüft, auch die Syntax nicht.

```
plot X, Y, C
```

`X`=X-Koordinate, `Y`=Y-Koordinate, `C`=Farbnummer `X, Y, C` müssen einfache numerische Variablen sein, (`A-Z`) keine Zahlen, keine Rechenausdrücke. Siehe auch Programmbeispiel "[Mandelbrot.PBAS](#)".

## poke / pO

```
poke "b", 32Bit-Adresse, 8Bit-Byte-Wert
poke "w", 32Bit-Adresse, 16Bit-Word-Wert
poke "l", 32Bit-Adresse, 32Bit-Long-Wert
poke "f", 32Bit-Adresse, 32Bit-Float-Wert
poke "x", 32Bit-Adresse, 64Bit-Float-Wert
poke "m", 32Bit-Adresse, 32Bit-(String-)Adresse, [Anzahl]

poke 32Bit-Adresse, 32Bit-Long-Wert      für 32Bit vereinfachte Syntax
                                            (statt: poke "l", ...)
```

```
ex. :
10 poke "b", @A$, 65 : poke "b", @A$+1, 0 : print A$
20 poke "f", @A, pi
30 poke "m", 0x10000000, @A$
```

Schreibt den Wert *Byte-Wert* (8-Bit) bzw. *Word-Wert* (16 Bit) bzw *Long* (32-Bit) oder *Float-Wert* (Fließkomma, 64Bit) in die RAM-Speicherzelle mit Adresse **32Bit-Adresse**. Es kann auch ein Speicherbereich (z.B. ein String) kopiert werden. Im Programm-Modus oder direkt. Im Beispiel wird in die Adressen von Variablen geschrieben, da diese Plätze *nicht riskant* sind – im Gegensatz zu vom System verwendetem Speicher. Wer den Prozessor kennt, kann gerne auch die Prozessor-Register umprogrammieren. Siehe auch [peek](#).

`poke "b"` kann seit V1.05A18 auch ins FlashPROM schreiben:

```
ex. :
poke "b", 0x000D0000, 131
```

allerdings nur Bytes und nur in den Sektor **10** (nur F405RG, F407VG und F407ZG mit 1Mbyte ROM). Es stehen 128KBytes zur freien Verfügung.

Adressbereich: 0x000C0000 - 0x000DFFFF

Das ist der Speicherbereich, der auch für evtl. Maschinenprogramme reserviert ist.

FlashPROM kann nicht wie RAM einfach ständig überschrieben werden. Bevor es neue Bytes aufnehmen kann, muss es gelöscht werden. Das geht nur sektorweise und geschieht mit [clear 10](#) (löscht die gesamten 128K) – das kann man ca. 10000 Mal wiederholen, dann ist es verschlossen. Als Variablen-Ersatz taugt es also nicht – dem fortgeschrittenen User fallen aber bestimmt interessante Möglichkeiten ein.

print, ? / pR

```
print Rechenausdruck,|;|\ String,|;|\ Funktion,|;|\ Variable ,|;  
?  
? A  
? not(A or (C and 255))
```

```
ex.:  
10 cls : locate 1, 1 : print "Hallo"; " "; "Thomas"  
20 let C$ = "pahlbasic V1.05" : print C$;
```

drückt den Inhalt von Variablen oder Textkonstanten oder Funktionen auf dem Terminal oder LCD aus. [print](#) rechnet auch Ausdrücke wie `A + (B * C)` aus. Ein Semikolon `:` am Ende der Anweisung unterdrückt den Zeilenvorschub. (Auf dem Terminal). Oder hängt die Komponenten ohne Zwischenraum aneinander. Ein Komma `,` setzt den Tabulator auf die nächste Position (für Tabellen). Der Backslash `\` fügt eine Zeilenschaltung ein. Siehe auch [lcd](#). Siehe auch [String Verkettung](#).

```
ex.:  
10 print " x", "sin(x)", "cos(x)"  
20 for N=0 to 10 step pi/10 : print N, sin(N), cos(N) : next N  
30 print A \ B \ C
```

Programm- oder Befehlsmodus. [print](#) funktioniert nicht in Interrupt-Routinen. Die Abkürzung '`?`' gilt nur für das Kommando (`pR` im Programm/Repl).

`print #`

schreibt in die seriellen Schnittstellen 1...4, in den USB, aufs LCD, in Dateien.

`print # 1, A$` schreibt in die serielle Schnittstelle Nr.1  
`print # 2, A$` schreibt in die MIDI-Schnittstelle, die aber auch für andere Anwendungen als MIDI genutzt werden kann.  
`print # 3, A$` schreibt in die GPS-Schnittstelle, die aber auch für andere Anwendungen als GPS genutzt werden kann.  
`print # 4, A$` schreibt in die serielle Schnittstelle Nr.4  
`print # 5, A$` schreibt in die USB-Schnittstelle  
`print # 9, A$` schreibt auf das LCD (ohne Umleitung mit `lcd1`)  
`print #10, A$` schreibt in die Datei mit Filehandler 10 (Filehandler=10...24)

`print #` verhält sich ähnlich wie 'normales' `print`, nur der Zeilenvorschub / Wagenrücklauf ist erst *nach* der Anweisung `print #` (und nicht *vorher* wie bei `print`), `print#` sendet auch keine Farbinformationen, Komma und Semikolon schalten den Vorschub aus. Das Komma fügt Tabulator ein. `print #` funktioniert nicht in Interrupt-Routinen.

Ausgabe auf der seriellen Konsole (ahmt 'normales' `print` nach):

```
print #1, chr$(13); chr$(10); "Hallo";
```

schließt man stattdessen ein Terminal an die MIDI-Schnittstelle an, dann:

```
print #2, "Hallo"
```

Bei den Schnittstellen 2, 3 & 4 bitte die Erfordernisse der angeschlossenen Geräte beachten. Es handelt sich um RS232-Schnittstellen mit quasi TTL-Level. Echte RS232-Schnittstellen erhält man, wenn man entsprechende Pegelwandler anschließt z.B. mit dem [MAX3232](#).

Die Schnittstellen 2, 3 und 4 werden nach dem Reset z.Z. mit 9600 Baud gestartet (konfigurierbar mit [config](#)).

`print # 9` schreibt direkt auf das LCD. Eine Umleitung mit `lcd1` ist nicht nötig. Es gilt das unter `lcd` im Zusammenhang mit `print` Gesagte.

`print #10...24 *` schreibt in eine geöffnete Datei siehe [fn Open\(\)](#). `print #` verhält sich dabei ähnlich wie `print` auf dem Terminal-Screen. Der Ausdruck von `print` wird in die Datei umgeleitet.

\*) Filehandler 10...20 für den Flash-Chip, 21...24 für die SD-Card.

## proc

### proc Labelname

Ab Version 1.05 gibt es Programm-Label. Sprungziele können jetzt Namen haben.

```
ex.:
10 proc Abfrage_Eingang1                                das Label
20 print diginp.0; chr(13);
30 wait 100
40 goto "Abfrage_Eingang1"                            Sprung zum Label
```

**proc** kennzeichnet eine Programm-Sprungmarke. Es kann überall im Programm stehen, aber stets am Zeilenanfang. **proc** wird von den Sprunganweisungen **break**, **goto**, **gosub** als Ziel erkannt. In der Sprunganweisung muss der **Labelname** in Anführungszeichen gesetzt werden. Er darf nur aus Buchstaben, Zahlen und dem Unterstrich bestehen. **Labelname** muss mindestens 2 Zeichen lang sein und mit einem Buchstaben beginnen. Keine weiteren Anweisungen in der **proc**-Zeile.

## put

### put Wert1 [ , Wert2, Wert3, ...]

sendet ein einzelnes (oder mehrere) Zeichen mit dem **Bytewert Wertx** über die Konsolen-Schnittstelle (Nr.1 oder USB). In **put** steckt nicht die Komplexität von **print**. **Wertx** darf auch **Null** sein! **Wertx** wird nicht geprüft, sondern auf Byte-Format zurechtgestutzt. Maximal 32 **Werte** (Zeichen). In Interrupt-Routinen sendet **put** nur 1 Zeichen.

```
ex.:
10 put 27, 91, 50, 74, 27, 91, 72                      VT100 Sequenz für cls
```

## put #

### put # Schnittstellen-Nummer, Wert1 [ , Wert2, Wert3, ...] put # Filehandler, Wert1 [ , Wert2, Wert3, ...]

**put #1** schreibt in die serielle Schnittstelle Nr.1  
**put #2** schreibt in die MIDI-Schnittstelle, die aber auch für andere Anwendungen als MIDI genutzt werden kann.  
**put #3** schreibt in die GPS-Schnittstelle, die aber auch für andere Anwendungen als GPS genutzt werden kann.  
**put #4** schreibt in die serielle Schnittstelle Nr.4  
**put #5** schreibt in die USB-Schnittstelle.  
**put #9** schreibt auf das TFT-LCD  
**put #12** schreibt in geöffnete Datei mit Filehandler 12      Filehandler=10...24

```
ex.:
10 put #1, 27, 91, 50, 74, 27, 91, 72                  VT100 Sequenz für cls
```

Maximal 31 **Werte** (Zeichen), sonst wie normales **put**. Die Schnittstellen **2**, **3** und **4** werden nach dem Reset z.Z. mit 9600 Baud gestartet. Siehe auch [config](#).

Siehe auch [fn Open\(\)](#). Es ist möglich sich eigene Dateiroutinen auf Byte-Ebene zu schreiben. **put#10** schreibt Bytes ohne Zwischenraum oder Trennzeichen in die Datei. Gelesen werden diese Zeichen mit [get#](#) - der perfekte Gegenspieler. Vielleicht schreibt ja jemand einen Datei-Hex-Editor?  
☺

ram, rom

schaltet für die Kommandos `jmp`, `goto`, `gosub`, `read` und den Adressenoperator `@` in das RAM bzw. in das ROM um. `ram` oder `rom` müssen vorher ausgeführt werden (nicht mit F401).

rcall

Siehe `call`.

rd

(remove Directory)

`rd "Files"`

entfernt das Unterverzeichnis `Files` von der SD-Karte, das nicht leer sein muss. Kann auch Dateien löschen.

read

`read Numerische Variable` (auch: System-Variable oder Array-Element)  
`read Stringvariable` (auch: Text-Array-Element)  
`read # Filehandler, Numerische Variable`

```
ex.:
10 read A(0), A(1), A(2), pwm1, B$, W
20 print A(0), A(1), A(2), pwm1, B$, W
30 read X, Y, Z, T(9)
40 print X, Y, Z, T(9)

100 data 102.5, 113.7, 114.44, 512, "pahlbasic"
200 data 106.4, 120.3, 133.5, 640, "Software"
```

`read` liest numerische Werte oder Zeichenketten aus `data`-Zeilen. Nach jeder `read`-Instruktion wird der Lesezeiger auf das nächste `data` Element gesetzt. Siehe auch `data`. Der `Variabletyp` muss mit dem `Konstantentyp` übereinstimmen. Mit `restore` kann man die Position des Lesezeigers selbst bestimmen.

Prozeduren und Funktionen haben einen eigenen Lesezeiger. Bei Start der Prozedur / Funktion wird dieser Zeiger auf die erste `data`-Zeile in der Prozedur / Funktion gesetzt. `read` liest innerhalb von Prozeduren und Funktionen auch nur bis zur `end`-Anweisung. Prozeduren und Funktionen sollten daher ihre eigenen `data`-Zeilen haben:

```
ex.:
10 rem data-Test                                     Hauptprogramm
20   restore "Daten"
30   read A, B, C : print A, B, C                  Aufruf der Prozedur
40   call Test()
50   read D, E, F : print D, E, F                  Hauptprogramm Ende
90 end.
95
1000 sub Test                                         die Prozedur Test
1020   read A, B, C
1030   print A, B, C
1040   data 55, 66, 77                                Prozedur-Daten
1090 end                                              Prozedur Ende
1100
2000 proc Daten                                      Daten für Hauptprogramm
2010 data 11, 12, 13, 14, 15, 16
```

```
> run
11      12      13
55      66      77
14      15      16
ready
```

Neue zusätzliche Syntax: `read # Filehandler, Numerische Variable`

Mit `read #` werden die mit `write#` gepackten Zahlen aus Dateien gelesen.  
Siehe [fn Open\(\)](#).

## receive / reC

für die Übertragung von Text-Dateien (Basic-Programm-Texte) vom PC zum Basic-Computer mittels [TeraTerm](#).

\*

Die Anweisung `receive` wartet auf den Zeichenstrom und setzt die empfangenen Zeichen zu einem Programm zusammen. `receive` führt vorher die Anweisung `new` aus. Der genaue Ablauf ist im Anhang "[Datei senden](#)" beschrieben.

Wurde die Konsole auf Schnittstelle 9 umgeleitet, muss der PC an Schnittstelle 1 angeschlossen sein.

<pre>&gt; receive</pre>	<b>Eingeben + Return-Taste</b>
<p>Antwort:</p> <pre>waiting for data &gt;      jetzt "Datei senden" im TeraTerm starten Transmission success. Program Lines: 12 ready &gt; -</pre>	

Ich habe vor kurzem von einem sehr engagierten User folgenden Tipp bekommen:  
schreibt man in die erste Zeile der Basic-Text-Datei nur die Zeichenfolge

`reC` `(und Return)`

also den verkürzten Aufruf der Anweisung `receive`, muss man die Anweisung gar nicht mehr im Direktmodus eingeben, sondern kann direkt aus TeraTerm die Basic-Text-Datei senden. Die Datei startet dann den Befehl selbst. Danke für den Tipp.

### Für Profis

`receive` kann aber auch zum Einfügen von Programmteilen und Zusammensetzen von Programmen eingesetzt werden. Dafür gibt es:

\*

`receive [Zeilen-Nummer]`

Die zu *empfangenden Dateien* dürfen hierbei absolut **keine** Zeilennummern enthalten weder am Zeilenanfang noch in `goto`, `gosub`, `break` oder anderen Anweisungen, stattdessen nur `Labels` verwenden. Solche Dateien sollte man zur besseren Unterscheidung mit der Datei-Endung `.INC` (include) versehen – normale Basic-Programm-Texte sollten die Endung `.PBAS` (vorzugsweise) oder `.TXT` besitzen. Es ist möglich, eine Art Bibliothek aufzubauen, mit Programmteilen, die immer wieder verwendet werden können. `receive Zeilen-Nummer` führt kein `new` aus, damit das Laden und Zusammenfügen mehrerer Dateien möglich ist.

pahlbasic jetzt ohne Zeilen-Nummern? Natürlich nicht - `receive [Zeilen-Nummer]` fügt die Zeilennummern ein. Beginnend mit `Zeilen-Nummer` für die erste Zeile und dann in 10-er Schritten höher. `Zeilen-Nummer` darf nicht kleiner als 1000 sein. Man kann also mehrere Programmteile z.B. in 1000-er Blöcken laden.

Lässt man `Zeilen-Nummer` weg, setzt `receive` automatisch Start-Zeile 1000 ein, führt

dann aber `new` aus. In *Basic-Text-Dateien* sind Zeilen-Nummern aber grundsätzlich verzichtbar.

Enthält die Datei trotzdem Zeilen-Nummern, werden diese von `receive Zeilen-Nummer` überschrieben (Fehlerzähler!). Das Programm läuft dann evtl. nicht mehr richtig.

Include- Dateien dürfen zusätzliche Kommentare enthalten, die mit `//` direkt am Zeilenanfang beginnen und nicht mitübertragen werden (nur ohne Zeilen-Nummern). Der Sende-Ablauf ist der gleiche wie oben beschrieben:

```
> receive 1000          Eingeben + Return-Taste
```

Enthält der Basic-Text `include`-Anweisungen, so werden Warnmeldungen ausgegeben. Die aufgeführten Dateien müssen dann manuell (z.B. mit `receive Zeilennummer`) angehängt werden, da `include` nur mit `load` von SD-Card funktioniert. Leere Zeilen werden nicht übertragen, mindestens ein Tabulator muss in der Zeile stehen.

- Der umgekehrte Weg: vom Basic-Computer in eine Text-Datei auf dem PC geht z.B. über ein **Listing** im Terminal-Fenster, markieren der Zeilen und ab in die Zwischenablage von Windows (Strg+c). Dann einfügen des Zwischenablage-Inhalts in ein Textprogramm. Für **Notepad++** gibt es Syntax-Dateien für pahlbasic (siehe rechts)

Oder: über die **Log-Funktion** von TeraTerm (**Datei \ Log...** starten dann **list** und danach **Datei \ Stop Logging**) und Verwenden der Log-Datei (nach Bearbeitung) als Basic-Text. Seit kurzem geht natürlich auch der Daten-Austausch via SD-Card.

```
proc Anzahl_Satelliten
    * GPS-Modul angeschlossen? GPS-Parsing eingeschaltet?
    print "Anzahl Satelliten: "; fn Satelites()
end.

function Komma(A, D) : 'A=Anzahl Kommas, D=ZeichenOffset
    let B = length(@AS) : let E = 1
    while E<B
        if A$(E) = "," then inc C
        if C=A then exitdo
        inc E
    wend
    let result = E+D
end.

function Satelites()
    let AS = gpos$("GGA")
    let result = ((AS$(fn Komma(7, 1))-48)*10+(AS$(fn Komma(7, 2)))-48
end.
```

\*

`receive "m"`

in Entwicklung

kann kurze Maschinensprache-Programme (als Intel-Hex-Dateien) laden. Die Maschinenprogramme starten an festen Adressen (0x00060000, 0x00064000, 0x00068000 und 0x0006C000 im Sector 7). Diese Adressen müssen natürlich in der Intel-Hex-Datei auch so stehen, sonst bricht `receive "m"` mit Fehlermeldung ab. Sie werden mit der Anweisung `user Programm-Nr` aufgerufen. Diese Programme werden dauerhaft im ROM gespeichert und stehen jederzeit wieder zur Verfügung. Weitere Infos auf Anfrage.

record

record "filename" [, SampleRate]

zeichnet eine Tonaufnahme auf SD-Card auf. `filename` muss die DateiEndung `.WAV` haben. Der AD-Converter im STM32F405/7 wandelt mit 12Bit. Der Signal/Rauschabstand ist mit Kassettenrecordern vergleichbar. Gespeichert wird die Aufnahme mit 16Bit (mono). `SampleRate` darf die Werte 11025, 22050 oder 44100 (bevorzugt) evtl. auch 48000 annehmen. Ohne Angabe wird mit 44100Hz gesampelt. Das Aufnahmesignal am AnalogPin sollte ca. 1V eff. (knapp 3Vpp) betragen. Das Signal sollte einen Gleispannungsoffset von 1.6V aufweisen (z.B. mit Spannungsteiler). Die Aufnahme wird durch Drücken einer Taste auf der Konsole beendet. Das

Wave-File kann auch am PC abgespielt werden.

[play](#) und [record](#) schalten auf höchste SPI-Geschwindigkeit. Das funktioniert nicht bei allen SPI-Displays, die ja auf dem gleichen Anschluss liegen (insbesondere nicht bei ILI9341). Mit ILI9488 funktioniert es garantiert - kurze Anschlussdrähte vorausgesetzt. Anschluss siehe [Anhang](#).

rem, '

remark (für Kommentare im Programmtext)

```
ex. :  
20 rem Messwertausgabe  
30 let A = 2*pi*pi : 'auch ein Kommentar
```

die Anweisung **rem** oder das **Hochkomma '** tun nix ☺. Sie dienen nur der Dokumentation im Programmtext und werden mit [list](#) angezeigt. Kommentare sollten großzügig eingesetzt werden, weil Vintage-Basic wenige 'sprechende' Namen besitzt.

Stehen **rem** oder '**'** nicht direkt hinter der Zeilen-Nummer, müssen sie durch den Doppelpunkt abgetrennt werden (auch wenn sie nichts bewirken, sind sie formal Anweisungen). Jeglicher Text hinter **rem** oder '**'** gilt als Kommentar. Deutsche Umlaute im Kommentar sind kein Problem, fügt man ein Anführungszeichen direkt hinter '**'** oder **rem** ein.

rename / reN

rename "alter Datei-Name", "neuer Datei-Name" [, Medium]

Benennt die Datei um. **rename** schert sich nicht um die Art der Datei, sondern tauscht nur den Namenstext aus. **Medium:** 1=Flash-Chip, 2=SD-Card

reset

die Anweisung **reset** ist eine Universalanweisung zum rücksetzen verschiedener Komponenten. Programm- und Direktmodus. z.Z. sind folgende implementiert:

\* **reset relayx1, x2, x3, ...** (**xN** = 1....16)

```
reset relay1, 4, 6, 9
```

Schaltet Relais 1, 4, 6 und 9 gleichzeitig (!) aus. Programm und Direktmodus. Bitte beachten: die Relais-Nr. ist um 1 höher als die Bit-Nr. der Systemvariablen **relay**. Die 'Relais'ausgänge können natürlich auch mit anderen Aktoren als Relais beschaltet werden. Auch als Chip-Select oder Impulsgeber können die Ausgänge dienen. Die Bezeichnung **relay** stammt noch von der AVR-Version, dort waren halt Relais angeschlossen. Siehe auch [config set](#) und negative 'Relais'-Logik.

Setzt man einen STM32F407ZGT ein, gibt es auch die Ausgänge 17...48 auf den Ports F und G. Allerdings asynchron und ohne Berücksichtigung von positiver oder negativer Logik. Auch der Resetzustand ist nicht konfigurierbar.

\* **reset config**

Setzt alle Einstellungen auf den Auslieferungszustand zurück **und führt einen Warmstart durch**. Es muss vorher eine Überschreib-Warnung mit "y" beantwortet werden.

\* **reset diginp6**

Digitaleingang 6 wird wieder zum Digitaleingang. Siehe auch [set diginp6](#).

- \* **reset impuls Impulskanal** (Impulskanal 1...4)  
Setzt den Impulszähler des **Impulskanals** zurück. Der mit impuls relay oder on impuls gestartete Impuls läuft ohne Unterbrechung abermals los.
- \* **reset intrpt**  
Schaltet **alle** mit **on** und **sound** aktivierten Interrupts und die zugehörige Hardware auf einmal aus. Sie müssen mit **on** oder **sound** wieder neu gestartet werden. Siehe auch: set intrpt für lediglich temporäres de-/aktivieren.
- \* **reset sound**  
schaltet den Sound ganz aus und gibt die Wandler Ausgänge wieder frei.
- \* **reset Variable.Bit** oder **reset Adresse, Bit**  
setzt das **Bit (0...31)** in der Variablen (**A ...Z**, Arrays) / bzw. Adresse zurück.  
Bitte Punkt bzw. Komma beachten.

**reset #**

Setzt einen einzelnen Ausgang auf einem Modbus-Modul zurück

**reset # Interface, Adresse, Coil-Nummer**

**Interface:** 1...4,   **Adresse:** 1...32 \*) , **Coil-Nummer:** 1...32                     \*) F407ZG: 1...240

setzt den Ausgang **Coil-Nummer** auf dem Modbus-Modul mit **Adresse** an der seriellen Schnittstelle **Interface** zurück. Es wird die Modbusfunktion 05 eingesetzt. Siehe auch set # und Anhang.

**ex. :**  
**10 reset #4, 21, 17**

Ausgang **17** auf dem Modbus-Modul mit Adresse **21** an serieller Schnittstelle **4** wird zurückgesetzt. Es können die Schnittstellen 1...4 verwendet werden, jedoch nur *eine* Schnittstelle für Modbus. (Es gibt nur einen RTS-Anschluss für die Richtungssteuerung \*\*) der Kommunikation – Modbuskommunikation mit den von mir eingesetzten Modulen der chinesischen Firma eletechsup ist halbduplex mit 9600 Baud über *eine* 2-Draht-Leitung).

\*\*) gilt nicht für das F407ZG-Board.

## restore / reS

```
restore [ Zeilennummer | Label | Adresse | #Filehandler [, Position] ]  
restore # Filehandler [, Position]
```

restore 100 setzt den Lesezeiger von `read` auf Zeile 100. Siehe [data](#), [read](#). Enthält Zeile 100 keine `datas`, wird mit Fehlermeldung abgebrochen.

```
ex.:  
10 restore 100                                     Lesezeiger auf Zeile 100  
20 read A, B  
30 restore                                         Lesezeiger auf 1. data-Zeile  
40 read A$, B$, C$  
50 print A, B, A$, B$, C$  
60 end.  
. . .  
90 data "hier", "auch", "datas"  
100 data 4711, 2021
```

```
> run  
4711 2021 hier auch datas  
ready
```

restore "Daten2" setzt den Lesezeiger auf die nächste `data`-Zeile, die dem `Label` `Daten2` folgt.

```
ex.:  
10 restore "Daten2"                                Lesezeiger auf Label  
20 read A, B  
30 restore                                         Lesezeiger auf 1. data-Zeile  
40 read A$, B$, C$  
50 print A, B, A$, B$, C$  
60 end.  
. . .  
90 data "hier", "auch", "datas"  
100 proc Daten2  
110 ' es folgen Daten  
120 data 4711, 2021
```

Ohne Angabe von `Zeilennummer` oder `Label` setzt `restore` den Lesezeiger wieder auf die *erste* `data`-Zeile im Programm. Eine Möglichkeit, den Data-Zeiger zwischenzuspeichern ist die Systemvariable [dataptr](#).

Seit V105A30 kann `restore #` auch den Lesezeiger einer geöffneten Datei auf dem Flash zurücksetzen (auf Anfang) z.B.: `restore #A` (vorausgesetzt, die Datei mit Filehandler `A` ist zum Lesen geöffnet). Man kann die Daten dann wiederholt einlesen. Siehe auch [fn Open\(\)](#).

Für experimentierfreudige User gibt es: `restore #A, Position` - setzt den Lesezeiger auf beliebige (Byte)-Position: `restore #A, 1222`

Bei SD-Karten kann sowohl der *Schreib-* als auch der *Lesezeiger* auf eine beliebige (Byte)-Position gesetzt werden.

## resume

```
resume [ Zeilennummer | Label | erl ]
```

wie `return`. Jedoch Rückkehr von einer [on\\_err\\_gosub](#) Fehlerbehandlungs-Routine.

Mit `resume` wird hinter die Fehlerstelle zurückgekehrt, mit `resume erl` zur Zeile mit dem Fehler, mit `resume Zeilen-Nummer` oder `resume Label` zu einer beliebigen Programm-Zeile.

return / reT

**return** [ Zeilennummer | Label ]

Rückkehr vom Unterprogramm. Es wurde zuvor mit [gosub](#) in ein Unterprogramm verzweigt. Mit **return** wird an die Stelle zurückgekehrt, die dem [gosub](#)-Aufruf folgt. Ist die [Zeilennummer](#) oder das [Label](#) angegeben, wird dorthin gesprungen und die mit [gosub](#) gespeicherte [Rücksprungadresse](#) verworfen.

```
ex.:
10 rem Hauptprogramm
20 gosub 100
30 Anweisung1
...
100 rem Unterprogramm
110   AnweisungX
120   AnweisungY
...
150 return
```

Aufruf des Unterprogramms  
Hierhin Rückkehr von return

Unterprogramm-Start

Rücksprung (zur Zeile 30)

rload

Siehe [load](#).

run, rrun

\*

**run**  
run "Programmname" [, Medium]  
run Programmnummer

Medium: 1=Flash-Chip, 2=SD-Card  
nur Flash-Chip

> run

startet das im RAM-Speicher befindliche Programm. **run** löscht vorher alle Variablen wie [clear](#).

> run "Alarmanlage.PBAS", 2

Lädt das Programm "Alarmanlage.PBAS" von der SD-Card in den Speicher und startet es.

> run 3

Lädt das Programm Nr. 3 vom Flash-Chip in den RAM-Speicher und startet es. Die Programm-Nummer mit [dir](#) ermitteln. **run** nur im Befehlsmodus.

\*

**rrun** ist exakt wie **run**, jedoch wird das Programm direkt im [ROM-Speicher](#) (ohne [load](#)) gestartet. Das Programm wurde zuvor z.B. mit [save@](#) oder [rload](#) im ROM gespeichert. **rrun** schaltet den Kommando-Modus für nachfolgende Anweisungen wie [jmp](#) oder [goto](#) auf das ROM um.

save / sA

save	speichert Programm nach Bearbeitung
save 0	ersetzt das Speichern im int. EEPROM beim AVR
save 1	ersetzt das Speichern im int. EEPROM beim AVR mit Autostart
save "Name" [, Medium]	speichert Programm mit Namen auf Medium Medium: 1=Flash, 2oder3=SD-Card
save @ [1]	speichert Programm im intern. ROM [mit Autostart]
save !	speichert Einstellungen

> save 0

Speichert das im RAM befindliche Programm dauerhaft im seriellen Flash. Es muss mit run gestartet werden. Die Datei 0 ist unsichtbar (AVR-EEPROM-Ersatz). Das Programm 0 wird bei Start des Computers automatisch geladen.

Ist kein serieller Flashspeicher vorhanden, wird im internen ROM gespeichert. (z.B. beim F411), dies lässt sich mit config save auch bei vorhandenem seriellen Flash erzwingen. \*)

> save 1

Wie save 0. Das Programm 0 wird nach einem Reset sofort gestartet. Ist kein Programm im RAM-Speicher wird eine Fehlermeldung ausgegeben.

Ist kein serieller Flashspeicher vorhanden, wird im internen ROM gespeichert. (z.B. beim F411), dies lässt sich mit config save auch bei vorhandenem seriellen Flash erzwingen. \*)

\*) Im internen ROM werden zusätzlich die Einstellungen gespeichert.

> save @

speichert im internen ROM

Das Programm (und die Einstellungen) wird – unabhängig von Einstellungen mit config – im internen ROM gespeichert. Nur für Prozessoren mit 1Mbyte Flash. Programme dürfen max 64K gross sein.

Seit kurzem können Programme auch im ROM laufen. Eine neue Anwendung ist das Speichern einer Sammlung von Funktionen und Prozeduren, die allen RAM-Programmen zur Verfügung stehen und den knappen Speicherplatz im RAM erweitern.

> save !

speichert Einstellungen

Es werden *nur* die config-Einstellungen im internen ROM gespeichert, ein dort evtl. vorhandenes Programm wird (nach Rückfrage) gelöscht. Nur für Chips mit 1Mbyte.

> save "Temperatur Alarm", 1

im Flash-Chip

Das im RAM befindliche Programm wird unter dem Namen "Temperatur Alarm" im seriellen Flash gespeichert. Ist kein serieller Flash-ROM vorhanden, wird das Programm im internen ROM ohne Namen gespeichert.

> save "Tempalrm.PRG", 2

auf der SD-Card

Dasgleiche auf der SD-Card. Dateinamen bitte im 8.3-Format (8 Zeichen für den eigentlichen Namen & 3-4 Zeichen für die Endung. Längere Namen (20 Zeichen) werden aber auch unterstützt. Programme bitte immer mit der Endung .PRG oder .PBAS sowie .INC speichern. .PRG=Speicherabbild, .INC/.PBAS=Basic-Text (nur SD-Card).

save entfernt sämtliche Zeilen-Nummern aus .PBAS Dateien! Daher bitte nur Labels verwenden. Werden die Zeilen-Nummern dennoch benötigt, Medium=3 einstellen.

```
> save  
Program No. 29 Already Exists  
overwrite?
```

nach Bearbeitung

Wurde das Programm mit `load` geladen, kann es (z.B. nach Bearbeitung) mit `save` wieder gespeichert werden, der Name muss nicht wiederholt werden. Es wird eine Überschreibwarnung ausgegeben, die mit `y` (Yes) bestätigt werden muss.

Werden Einstellungen zusätzlich zum Programm gespeichert, werden diese beim Laden des Programms nicht automatisch mitgeladen. Dies muss mit `load !` explizit ausgeführt werden.

## seed

*nur für den STM32F411 und den STM32F401*

### seed Startwert

Setzt den **Startwert** (16-Bit Word-Wert) für die Systemvariable `rand` (Pseudo-Zufallszahl). Im Gegensatz zum STM32F405/7, der einen Hardware-Zufallsgenerator hat, müssen auf dem F411/F401 die Zufallszahlen mit einer Rechenoperation erzeugt werden. Es werden immer die gleichen Zahlen erzeugt. Daher kann man mit `seed` ein wenig Abwechslung in die Zahlenreihe bringen. Mehr Zufall ins Spiel bringt man, indem man als Startwert z.B. den Zählerstand des Millisekunden-Zählers einsetzt. `rand` auf dem F411/F401 liefert Pseudo-Zufallszahlen zwischen 0 und 31767 (0x7FFF). **Startwert** wird auf 16Bit zurecht gestutzt. Der STM32F405/7 kennt diese Anweisung nicht und meldet einen Syntax-Error.

Ein digitaler Würfel gefällig?

```
ex.:  
10 seed milli  
20 while inkey<>("e")  
30   print int(rand/0x7FFF*6)+1  
40   wait buf  
50 wend
```

'letzte Taste war ein 'e'? Programm beenden!  
'Würfelausgabe auf Terminal  
'beliebige Taste drücken für neuen Wurf

Ob die Verteilung der Pseudo-Zufallszahlen gleichmäßig ist, habe ich nicht überprüft. Wohl aber die 'echten' Zufallszahlen, die der F405/7 erzeugt. Das [Würfelbeispiel](#) für den F405/7 liefert sehr gerecht verteilte Würfelergebnisse (Überprüft mit 1000000 Würfen).

## select / sel

<b>select / case</b>	Auswahl / Fall-Entscheidung ("Programmschalter")
Der select/case Anweisungsblock:	
<b>select Ausdruck</b>	Startet den Anweisungsblock
<b>case Wert1, Wert2, Wert3, ...</b> Anweisungen...	werden ausgeführt, wenn in der Werteliste ein Wert mit Ausdruckswert übereinstimmt, die Liste braucht nur einen Wert
<b>case Wert1 to Wert2</b> Anweisungen...	werden ausgeführt, wenn im Wertebereich ein Wert mit Ausdruckswert übereinstimmt. Wert2 muss grösser als Wert1 sein.
<b>case &gt; Wert</b> Anweisungen...	werden ausgeführt, wenn der Wertevergleich wahr ist (wie bei "if")
<b>case &lt; Wert</b> Anweisungen...	werden ausgeführt, wenn der Wertevergleich wahr ist (wie bei "if")
<b>celse</b> Anweisungen...	werden ausgeführt, wenn keiner der obigen Fälle aufgetreten ist
<b>selend</b>	beschließt den Block

```
ex.:
100 select $Auswahl
110   case 5
120     set relay 5
130     wait 100
140     reset relay 5
150   case 19 to 22 : gosub 3000      Werteliste hat nur einen Wert: 5
160     print "OK"                   Zeile 120-140 ausführen,
170   case >16                      wenn case 5 ($Auswahl=5) wahr ist
180     print "Relay 17-18 gibt es nicht"
190     print "bitte wiederholen"      Bereich von Fällen
200   case 8, 9, 12                  überlappt mit Zeile 150
210     set relay 1
220   case <3
230     gosub 1000
240   celse
250     print "falsche Eingabe"      mehrere Fälle, gleiche Aktion
290 selend                         Wertevergleich
                                    alle übrigen Werte von A
                                    selend steht für select end
```

Die Programmausführung hängt hier (nur) von der Variablen **\$Auswahl** ab. Die Anzahl der **case**-Fälle ist nicht beschränkt. **Ausdruck** wird ausgerechnet und das Ergebnis für exakte Vergleiche in eine 32Bit Integerzahl gewandelt. (Wertebereich: -2147483648 ... 2147483647)

**celse** darf nur *einmal* und zwar als *letzter* Fall vorkommen, muss aber nicht.

Die **selend**-Zeile muss die letzte Zeile im **select**-Block sein und darf nicht fehlen, sie enthält keine Anweisungen!

Überlappen sich die Bedingungen, wird die **case**-Zeile ausgeführt, die in der Reihenfolge vorher „dran“ ist.

Die Anweisungen `select`, `case`, `celse`, `selend` müssen jeweils am Zeilenanfang stehen (Einrückungen zählen nicht).

In Basic wird der `select`-Block verlassen, sobald ein `case`-Fall oder der `celse`-Fall eingetreten ist (anders als `switch` in C, wo noch ein `break` hinzugefügt werden müsste).

Das Beispiel zeigt, dass es auch schnell unübersichtlich werden kann, aber im Gegensatz zu [on gosub](#), sieht man was passiert.

`select/case` kann auch verschachtelt werden. Der innere Block muss beendet sein, bevor der äussere Block weiterläuft:

```
10 select x                                äusserer Block
20   case 1 : AnweisungX1
30   case 2 : AnweisungX2
40     select y                            innerer Block nur, wenn X = 2
50       case 1 : AnweisungY1
60       case 2 : AnweisungY2
70       case 3 : AnweisungY3
80     selend
90   case 3 : AnweisungX3
100 selend
```

Für einen "sauberen Abschluss" des Programm-Blocks, keinesfalls mit `goto` "ausbrechen".

Noch' n Gedicht:

```
10 cls
20 input "Artikel? ", A$
30 select A$(1, 3)
40   case "der" : print " ist männlich"
50   case "die" : print " ist weiblich"
60   case "das" : print " ist sächlich"
70 selend
```

So funktioniert `select/case` mit Strings.

`selend`

steht für `select end`. siehe [select](#).

`send`

Sendet Daten über den RS485-Modbus.

```
send # Interface, Adresse, Funktion, Parameter1, Parameter2, ...
send # Interface, Adresse, Funktion, String
```

`Interface`: 1...4, `Adresse`: 1...255, `Funktion`: 1,2,3,4,5,6,15,16, `Parameter`: 0...255

Direkte Kommunikation mit Steuermodulen via Modbus. Siehe [Anhang](#).

`Interface` ist die serielle Schnittstelle, an der der Bus angeschlossen ist. `Adresse` ist die Moduladresse, des Steuermoduls mit dem Daten ausgetauscht werden soll. `Parameter` gehören zur jeweiligen Modbus-`Funktion`, z.B. Adressen. Die Antwort des Moduls wird im Kommando-Modus auf dem Terminal im Hex-Zahlenformat ausgedruckt, im Programm steht die Antwort in `H$`. Die zweite Syntax ist für das einbetten von Basic-Instruktionen reserviert (`Funktionen` ab 100). Siehe `server.send` fügt die Zeichen für den `CRC` bereits ein!

## server

Verwandelt den Basic-Computer in ein Modbus-Modul. (nur STM32F405/7)

server # Interface, Adresse auf dem Slave starten  
z.B. server #4, 1 Interface: 1...4, Adresse 1...240

Standartmässig ist der Basic-Computer der Master in einem Modbus-Netz. Mit der Anweisung `server` ist es jedoch möglich, auch einen oder mehrere (weitere) Basic-Computer als Slave zu betreiben. Nach dem Start der Anweisung `server` hört der Basic-Computer nur noch auf 'Signale' auf dem RS485-Bus. Er versteht 'normale' Modbus-Kommandos (Ein-/Ausgänge setzen und abfragen) sowie Basic-Anweisungen, die in das Modbus-Format eingepackt wurden.

- \* Basic-Anweisungen werden (auf dem Master) z.B. so gesendet:

send #4, 1, 100, "print , 'Hallo Welt' " + chr\$(13) Der Slave schickt das Kommando 1:1 zurück

oder so:

```
let A$ = "print 'Temperatur: ', T" + chr$(13)           chr$(13) 'drückt' die Return-Taste
send #4. 1. 100. A$
```

oder so:

`send #4, 1, 100, "rcall A1" + chr$(13)` ruft die Prozedur A1 auf dem Slave (im ROM) auf

**send #4, 1, 102, "A%"** fragt die numerische Variable A ab (A...H möglich)  
**send #4, 1, 102, "A\$"** fragt die String-Variable A\$ ab (A\$...H\$ möglich)

Funktion 102 beantwortet der Slave mit einem Modbus-String, bestehend aus:  
Adresse, Funktion, Anzahl Bytes, Byte1, Byte2, Byte3, ..., 00, CRC1, CRC2 im Hexformat

z.B.: 01 66 03 39 39 00 CB DB      Adresse 1, Funktion 66 (102 dez), 3Bytes: „99“, Zahlen als String, 0 markiert das Ende der Antwort – innerhalb eines Programms (auf dem Master) wird die Antwort in H\$ gespeichert, dessen Zeichen ja auch als Dezimalwerte (ASCII) ausgewertet werden können. Die Interpretation obliegt dem User.

- \* Normale Modbus-Kommandos funktionieren genauso, wie für die Module:

**set #4, 1, 2** schaltet (Relay)-Ausgang **2** auf dem Slave 1 ein  
**get input #4, 1, 10, A** fragt den Zustand von dig. Eingang **10** ab  
**get ad3 #4, 1, A** fragt den analogen Eingang **3** ab

Anführungszeichen innerhalb von Stringliteralen in Anführungszeichen kann pahlbasic nicht. Daher für die inneren Anführungszeichen Hochkommas schreiben (gilt nur hier).

Die Funktionen ab **100** sind (von mir) für Basic-Anweisungen innerhalb von Modbus-Kommandos vorgesehen. Damit die Kommandos nicht zu lang und zuviele werden, ist es sinnvoll, Prozeduren oder Funktionen auf dem Slave-Computer vorzuhalten, die die eigentliche Arbeit machen (z.B. Temperaturen oder Gebäude-Grundrisse anzeigen, Tonausgabe anstoßen). Der Master muss dann diese Funktionen und Prozeduren (aus der Ferne) nur noch mit **call/rcall**-Anweisungen - eingepackt in die Modbus-Kommandos - via **send** aufrufen. Ein ganzes Programm kann im Slave-Modus nicht laufen. Die Anweisung **server** kann aber sehr wohl Bestandteil eines Programms (z.B. mit Autostart) sein, das dann bei Aufruf von **server stoppt**. Dieses Programm enthält auch die nötigen Prozeduren und Funktionen. Es wird z.B. mit **save@1** im ROM gespeichert oder aber von SD-Card geladen. Während des Ablaufs der Prozeduren /Funktionen kann der Basic-Computer nicht auf Modbus-Kommandos reagieren, deshalb die Prozeduren oder Funktionen kurz halten.

Im Server-Modus gibt es Error-Meldungen (auf dem Slave) aber keinen Abbruch. Der Master erhält im Fehlerfall lediglich keine Bestätigung und meldet einen Hardware-Fehler. Beendet wird der Slave-Modus durch Druck auf den Reset-Taster oder mit **ctrl+c**.

## set

die Anweisung `set` ist eine Universal-Anweisung zum Setzen verschiedener Komponenten. Programm- und Direktmodus. z.Z. sind folgende implementiert:

- \* `set relayx1, x2, x3, ...` (xN = 1 ... 16)

```
set relay2, 4, 5, 9
```

Schaltet z.B. Relais 2, 4, 5 und 9 gleichzeitig (!) ein. Programm und Direktmodus. Bitte beachten: die Relais-Nr. ist um 1 höher als die Bit-Nr. der Systemvariablen `relay`. Die 'Relais'ausgänge können natürlich auch mit anderen Aktoren als Relais beschaltet werden oder als Chip-Select usw. eingesetzt werden. Siehe auch [config set](#) und negative 'Relais'-Logik.  
Setzt man einen STM32F407ZG ein, gibt es auch die Ausgänge 17...48 auf den Ports F und G. Allerdings asynchron und ohne Berücksichtigung von positiver oder negativer Logik. Auch der Resetzustand ist nicht konfigurierbar.

- \* `set clock Wert`

kalibriert die Systemuhr. Geht die Uhr nach, sollte `Wert` positiv sein, geht sie vor, helfen negative Werte. Schritte: +1 = ca. +1ppm -1 = ca. -1ppm

```
set clock -22  
set clock 18
```

verlangsamt die Uhr um ca. 22ppm  
beschleunigt die Uhr um ca. 18ppm

Die Werte kann man auch wieder auslesen mit `print sys(7)`. Max +/- 500ppm. 1 Sekunde pro Tag entspricht ca. 12ppm. Evtl. durch ausprobieren herantasten.

- \* `set diginp6 as fcount` 'Digital-Eingang 6 wird Frequenz-Zähler  
`set diginp6 as ictount1` 'Digital-Eingang 6 wird Impuls-Zähler  
`set diginp6 as ictount2` 'Digital-Eingang 6 wird Impulslängen-Zähler

Digitaleingang 6 wird mit dem internen Zähler 9 verbunden und zählt Impulse. Funktioniert nur mit Eingang 6, deshalb ist die Eingangs-Nr. nicht berechenbar, sondern muss als Ziffer '6' geschrieben werden:

<b>ex1.:</b>	<b>Frequenzmessung</b>
10 set diginp6 as fcount 20 print sys(3)	Eingang 6 wird zum Frequenz-Zähleingang Ausgabe der Frequenz in Hz
<b>ex2.:</b>	<b>Impulszählung</b>
10 set diginp6 as icount1 20 let A = sys(4) 30 wait 10000 40 print sys(4)	Eingang 6 wird zum Impuls-Zähleingang Dummy-Anweisung, setzt Zähler zurück beliebige Mess-Zeit abwarten Ausgabe der Impuls-Zahl, der Zähler wird zurückgesetzt
<b>ex3.:</b>	<b>Impulsmessung1</b> (wartend)
10 set diginp6 as icount2  20 print sys(34)	Eingang 6 wird zum Impuls-Längen-Zähleingang Ausgabe der Impulslänge in µS, sys(34) wartet auf den Impuls und sein Ende
<b>ex4.:</b>	<b>Impulsmessung2</b> (nicht wartend)
10 set diginp6 as icount2  20 let A = sys(4)  30 impuls relay 1, 1000  40 print sys(4)	Eingang 6 wird zum Impuls-Längen-Zähleingang Dummy-Anweisung, setzt Zähler zurück  Eingang 6 mit Relais 1 verbinden und Impuls von 1 Sekunde erzeugen gemessene Impulslänge in µS Ergebnis: 1000053

Die Zählfunktion kann bis in den MHz-Bereich arbeiten. Die Genauigkeit der Frequenz-Messung hängt vom Quarz des Basic-Computers ab. Typisch sind +/- 50ppm. Bei höheren Frequenzen (>1MHz) kann auch die Software-Latency eine Rolle spielen. Die Einstellung bleibt bis zum Reset oder der Anweisung reset diginp6.

Weil die Zählfunktion auf sehr schnelle Signale im Nano-Sekunden-Bereich reagiert, sind mechanische Kontakte als Geber ungeeignet. Die Impulse sollten von einer elektronischen Schaltung kommen.

#### \* **set gps=on      set gps=off**

Schaltet das GPS-parsing temporär ein / aus. Siehe auch [gps](#), [config gps](#).

#### \* **set input=Schnittstellen-Nummer**

Leitet die Konsole temporär auf die angegebene **Schnittstelle (1...9)** um. Schnittstellen-Nr. 0 stellt wieder zurück. Für Tests vielleicht nützlich. Es sollte natürlich ein Konsolengerät an der betreffenden Schnittstelle angeschlossen sein, sonst hilft der Reset-Taster ☺. Siehe auch [config input](#) für eine dauerhafte Einstellung.

Seit kurzem gibt es auch die Schnittstellen 6, 7 und 8 mit 'schrägen' Kombination von Ein- und Ausgabe:

<b>Schnittstelle</b>	<b>Input</b>	<b>Output</b>
1	Seriell 1	Seriell 1
2	Seriell 2	Seriell 2
3	Seriell 3	Seriell 3
4	Seriell 4	Seriell 4
5	USB-Device	USB-Device
6	USB-Keyboard	Seriell 1
7	Seriell 1	TFT-Display
8	USB-Device	TFT-Display
9	USB-Keyboard	TFT-Display

\* **set intrpt=off**      **set intrpt=on**

Deaktiviert alle **on**-Interrupts temporär (**off**) und aktiviert sie wieder (**on**). Siehe auch: [reset intrpt](#) für vollständiges Abschalten aller Interrupts.

\* **set lcd color=on**      **set lcd color=off**

Schaltet die Farbpalette temporär ein (**on**) oder aus (**off**). Momentanen Zustand abfragen mit [sys\(54\)](#).

\* **set print color=VT100-Farbnummer**

Stellt die Text-Farbe für alle folgenden **print**-Anweisungen temporär ein. (Also bis zum nächsten **set print color** oder Reset). Farbnummer **16** bewirkt, dass **print** keine Farbinformation sendet. Siehe auch [VT100 Farben](#).

**ex. :**

10 **set print color=12**

hellblau, Standart ist 7 (weiss)

\* **set sound [Phasenverschiebung, Faktor]**

Sorgt für einen einsatzbereiten Sound mit einfachen Grund-Einstellungen: Schaltet die DACs (Digital/Analog-Wandler) auf Soundausgabe um, füllt den Sample-Speicher (Array **S()**) mit einer Sinus-Welle und stellt mittlere Lautstärke ein. Der Hüllkurvengenerator erhält Werte für einen Ton mit Anschlag. Das Tempo wird auf 120 Beats/sec eingestellt. Benötigt ca 0.5 Sekunden zur Ausführung, funktioniert nicht in Interrupt-Routinen. Siehe auch [fn fourier](#) und [fn fmod](#). Siehe auch [Audio-Anschluss](#).

```
set sound  
play 70, 8, 73, 8, 76, 8, 79, 8
```

Werden zusätzlich die Parameter **Phasenverschiebung** und **Faktor** angegeben, verändert diese Einstellung den Ablauf der Anweisung **play**. **play** verbindet in diesem Fall den Hüllkurvengenerator fortlaufend mit der FM-Synthese und verändert so den Klang der Noten während des Spiels, was den Klang lebendiger macht. Nachteile: **play** wartet immer auf das Ende der Decay-Phase (ohne Parameter startet **play** lediglich die Note), deshalb ignoriert **play** in Interrupt-Routinen die Parameterangabe. **Faktor** darf Werte von **2...9** annehmen, **Phasenverschiebung < 65535**.

```
set sound 150, 5  
play 70, 8, 73, 8, 76, 8, 79, 8
```

\* **set Variable.Bit**      oder      **set Adresse, Bit**      (Bit: 0...31)

setzt das Bit mit der Nummer **Bit** im **Variable**wert oder dem 32-Bit-Wert an der **Adresse**. Bitte Punkt bzw. Komma beachten.

**set #**

Setzt einen oder mehrere Ausgänge auf Modbus-Modulen

```
set # Interface, Adresse, Coil-Nummer  
set # Interface, Adresse, Hi/Lo to 16-Bit-Wert  
set mod Bytes
```

Die erste Syntax schaltet den Ausgang **Coil-Nummer** auf dem Modul mit **Adresse** an der Schnittstelle **Interface** ein. (Vergleichbar mit **set relay x** für die internen Ausgänge des Basic-Computers - siehe oben). Löschen des Aus-

gangs mit `reset #`. Siehe auch [reset #](#) und [Anhang](#) für weitere Erläuterungen.

Die zweite Syntax schickt ein 16-Bitwort an den Ausgangsport des Moduls, was die 16 Ausgänge auf einen Schlag ändert (vergleichbar mit `let relay=X`).

Für Module mit 32 Ausgängen (oder die Ausgänge zählen von 17...32) gibt es den High/Low-Schalter, `Hi/Lo=1` wählt die obere Porthälfte mit den Ausgängen 17...32 oder `Hi/Lo=0` wählt die untere Porthälfte, Ausgänge 1...16. Nicht alle Module verstehen diese Anweisung.

Bit 0 auf der unteren Porthälfte schaltet Ausgang 1, Bit 0 auf der oberen Porthälfte schaltet Ausgang 17 etc. Der Ausgang wird eingeschaltet, ist das Bit gesetzt (Bit=1), der Ausgang wird ausgeschaltet, ist das Bit gelöscht (Bit=0).

Der Modbus ist hier auf com4 angeschlossen z.B. :

```
ex. :  
10 set #4, 3, 17          schaltet nur Ausgang 17 auf Modul 3 ein  
20 set #4, 3, 1 to 3      schaltet Ausgang 17 & 18 ein und Ausgänge 19...32 aus  
30 set #4, 3, 0 to 65535   schaltet die Ausgänge 1...16 ein (Bits 0...15)
```

Im Gegensatz dazu holt `get relay #4,3,4,A` den Zustand aller 32 Ausgänge in 4 Bytes auf einmal. Den Zustand eines einzelnen Ausgangs kann man dann mit dem Dezimalpunkt ermitteln z.B. mit: `print A.0` für Ausgang 1. Wenn man viele gleiche Module ansprechen und sich den 3. Parameter (4) sparen will (spart ja auch Übertragungszeit), kann man die Anzahl der Bytes, die nötig sind, um die Zustandsbits zu speichern, auch mit: `set mod Bytes` einstellen (gilt nur für `get input#/get relay#`, aktuelle Bytes abfragen mit `sys(126)`). Nach Reset: 4 Bytes.

## sleep

Versetzt die CPU in den Standby-Modus und verringert so die Stromaufnahme. Interessant für das DIY-More-CPU-Board ohne RTC-Batterie-Anschluss, das man nicht stromlos machen kann – ohne Uhrzeit, Datum und Einstellungen zu verlieren. Bei diesem Board muss die 'Uhrenbatterie' halt etwas 'dicker' sein, um die komplette CPU mit Strom versorgen. Z.B.: 3,7V Handy-Akku. Der Stromverbrauch der CPU sinkt von ca. 65...90mA auf ca. 6,5mA.

Aus dem Schlaf-Zustand erweckt die CPU nur ein Druck auf den Reset-Taster. Die automatische Sommerzeit-Umstellung wird deaktiviert. Siehe [Anhang](#).

## sort

```
sort Array-Startelement to Array-Endelement  
sort Array-Endelement to Array-Startelement
```

```
ex. :  
10 for N=0 to 19 : let A(N) = rand : next N  
20 'Array mit Zufallszahlen füllen  
30 sort A(0) to A(19)           aufwärts sortieren
```

Sortiert das angegebene Array aufwärts vom Start-Element bis zum End-Element.

Wenn das Array-Endelement zuerst angegeben wird, wird das Array abwärts sortiert:

```
ex. :  
10 for N=0 to 19 : let A(N) = rand : next N  
20 'Array mit Zufallszahlen füllen  
30 sort A(19) to A(0)           abwärts sortieren
```

## sound / sO

**sound** gibt einen Ton mit ganzzahliger Frequenz auf einem von 6 Tonkanälen (Stimme) aus. Der erzeugte Ton erklingt solange, bis er wieder ausgeschaltet wird. Alternativ gibt es die 'musikalische' Anweisung [play](#):

**sound Stimme , Frequenz**

sound 0, 100

man hört nichts

Im oberen Beispiel wird ein Tonsignalsignal auf Stimme 0 mit der Frequenz z.B. 100 Hz ausgegeben. Man hört aber nichts! Der Soundspeicher ist noch leer! Aber jetzt:

**ex.:**

```
10 'aber jetzt eine Tonleiter:  
20 let A = 440 : let volume = 50          Startton, Lautstärke setzen  
30 'Soundspeicher mit Sinusschwingung füllen, siehe Arrays:  
40 for N=0 to 1023 : let S(N) = sin(pi*N/512)*2047 : next N  
50 for N=1 to 12                          12 Halbtöne  
60   sound 0, A                           auf Stimme 0, Frequenz in A  
70   let A = A*(2^(1/12))                 nächsten Ton berechnen  
80   wait 500                            halbe Sekunde warten  
90 next N
```

Es gibt 6 Stimmen – 3 links (Stimmen 0, 2, 4) und 3 rechts (Stimmen 1, 3, 5). Jede Stimme kann einen Ton (also 6 gleichzeitig) abspielen. Bevor ein Ton-Signal ausgegeben werden kann, muss es erzeugt / geladen werden. Siehe auch [Array S\(\)](#).

Im unteren Beispiel füllt die Zeile 40 den Soundspeicher mit einem Sinussignal mit 1024 Samples (entsprechend 2\*Pi) sowie einer Amplitude von +/-2047.

Der Soundspeicher wird mit einer Samplefrequenz von 32768Hz abgespielt. Dieser Wert beeinflusst das Basic relativ wenig. Allerdings werden Töne mit höherer Frequenz mit wenigen Samples abgetastet. 1000Hz z.B. nur noch mit 32 Samples. Oberhalb der 3.Oktave nimmt die Klangqualität daher hörbar ab.

Bei zeitkritischen Aktionen (z.B. Zugriff auf den seriellen Flash-Programmspeicher) werden die Timer-Interrupts abgeschaltet, es knackt dann bei der Soundausgabe.

**volume** setzt die Lautstärke siehe [volume](#).

sound 1, 0

Frequenz 0 schaltet den Ton der Stimme 1 aus

Der erste Aufruf der Anweisung **sound** dauert länger (ca. 0,5 Sec) als die folgenden. Grund ist, dass der Digitalanalogwandler für die Soundausgabe auf einen mittleren Wert eingestellt werden muss. Dies würde einen lauten 'Knall' erzeugen. Daher werden die Ausgänge langsam „hochgefahren“. Einen kurzen Knacks (vom Einschalten der Pins) hört man trotzdem. Siehe auch Anhang [Audioanschluss](#).

Eine Dummy-Anweisung **sound 0, 0** am Programmstart kann der Vorbereitung der Soundausgabe dienen. Siehe auch "sound-Test" Beispiel-Programm. Alternativ gibt es die Anweisung [set sound](#), die alle nötigen Einstellungen für die erste Soundausgabe vornimmt.

Für bessere Soundausgabe müssten externe Sound-Chips angeschlossen werden. (VS1053 mit MP3 und MIDI ist geplant)

## spi

**spi Relay-Nr, Datenbyte [, Variable]**

Es werden Datenbytes über den SPI-Bus gesendet oder empfangen. Wird nur das Datenbyte angegeben, wird es über den SPI gesendet. Wird auch die Variable angegeben, wird ein Byte empfangen und in der Variablen gespeichert – das Datenbyte ist in diesem Fall nur ein Dummy.

Zusätzlich muss einer der (Relay)-Ausgänge (ohne Relais) als Chip-Select zur Verfügung gestellt werden. Siehe Anhang. (Das ist der Nachteil von SPI – sind die Ausgänge 'knapp' evtl. auf I2C ausweichen, einige Chips sind sowohl mit SPI- als auch I2C-Interface erhältlich):

<b>ex.:</b> 10 <b>spi relay8, 0, A</b>	empfängt ein Byte und übergibt es an A, es wird Relay8-Ausgang als Chip-Select verwendet
<b>ex.:</b> 10 let A = 2550 : let A &= 4095 20 <b>spi relay1, (A&lt;&lt;8) or 0x30</b>	sendet obere 4 Bit an MCP4921 (12Bit DAC)
30 <b>spi relay1, A and 0xFF</b>	sendet untere 8 Bit an MCP4921

Z.Z. können nur *Bytes* gesendet/empfangen werden. Für 16Bit bitte 2 Bytes senden/empfangen.

Auf dem 407mini-Board teilt man sich den SPI mit dem TFT-Display, der SD-Card und der SoundKarte. Wird der Bus nicht ordnungsgemäß freigegeben, sind Störungen sehr wahrscheinlich.

## stamp

**stamp** speichert einen Zeitstempel (aus Datum und Uhrzeit). Als String mit **stamp\$** wieder auslesbar. **stamp** macht nur innerhalb von Programmen Sinn, weil der Zeitstempel im Kommando-Modus ständig aktualisiert wird. Zeitstempel lassen sich in Variablen (**A-Z**) speichern. Siehe dazu auch: [sys\(66\) ... sys\(69\)](#).

<b>ex.:</b> 10 <b>stamp</b>	erzeugt System-Zeitstempel
20 <b>print stamp\$</b>	Ergebnis: Mo, 23.10.2023, 11:02:34
<b>ex.:</b> 10 <b>stamp A</b>	erzeugt System-Zeitstempel und speichert ihn in A
20 <b>print stamp\$(A)</b>	Ergebnis: Mo, 23.10.2023, 11:02:34

## start, !, rstart

- \* **start** startet Prozeduren, Funktionen und Interrupts. Im Programm ist **start** Bestandteil der on-Anweisung.

Im Direktmodus ist es ein Kommando zum Ausführen (Testen) von Prozeduren oder Funktionen – exakt wie call im Programm. Bei Funktionen wird zusätzlich der Funktionswert ausgedruckt. Die Abkürzung '**!**' gilt nur für das Kommando:

<b>! Fourier(7, 3, 5, 4, 3, 2, 1)</b>	startet eingebaute fn Funktion
---------------------------------------	--------------------------------

Setzt man den **Labelnamen** in Anführungszeichen, können auch Interrupt-Routinen getestet werden.

- \* **rstart** startet eine Prozedur direkt (ohne **load**) im internen ROM (nicht F401).

## sub

<b>sub Labelname</b>	ohne Variablenliste
<b>sub Labelname(Variable1, Variable2, ...)</b>	mit Variablenliste

Ab Version 1.05 gibt es Programm-Label. Sprungziele können jetzt Namen haben:

```
ex.:
10 sub Relais_Setzen($Bit, $Wert)           das Label mit Variablenliste
20   if ($Bit > 0) and ($Bit < 15)
30     let relay.$Bit = $Wert
40     wait 500
50   endif
60 end                                         Ende der sub-Routine
```

**sub** kennzeichnet eine Programm-Sprungmarke. Es kann überall im Programm stehen, aber stets am Zeilenanfang. **sub** ist ausschließlich für *Unterprogramme* reserviert.

Allgemeine Label schreibt man mit: [proc](#).

**Labelname** muss mindestens 2 Zeichen lang sein und mit einem Buchstaben beginnen. Er darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen. Keine weiteren Anweisungen in der **sub**-Zeile.

Das **sub**-Label leitet ein Unterprogramm (Prozedur) ein, an das die aufrufende Anweisung ([call](#)) Argumente übergeben kann. Die Argumenteliste (von [call](#)) wird eins zu eins in die Variablenliste (von **sub**) eingelesen (es werden die Werte übergeben – byval). Die Klammer darf weggelassen werden, wenn sie leer ist. Stringargumente (z.B. Literale in Anführungszeichen) sind auch möglich.

Das Unterprogramm endet mit der Anweisung **end**.

## text

**text [x, y, Modus,] Textstring**

gibt einen Text an der angegebenen Position **x, y** auf dem TFT-LCD pixelgenau aus. **Textstring** kann ein beliebiger in pahlbasic erlaubter **String** sein. **x** und **y** können auch ein Rechenausdruck sein. Die Zeichenfarbe wird mit [color](#) eingestellt. Es kann aus **8 Fonts** gewählt werden. ([font 0 ...font 7](#)). Die Textorientierung steckt in den Fontnummern. Fonts 8...15 und Fonts 16...23 sind Wiederholungen der Fonts 0...7 aber mit vertikaler Orientierung. **Modus** kann die Werte **1..8** oder **0** annehmen. **Modus=1** erlaubt den Ausdruck von sich ändernden Strings (z.B. Uhrzeit). Dieser **Modus** verwendet **H\$** als Zwischenspeicher. (**Modus=2** nutzt **G\$**, **Modus=3** nutzt **F\$** usw...). Siehe Programmbeispiel [Digital\\_Clock2.PBAS](#). **Modus=0** oder einfach weglassen bedeutet normale Textausgabe ohne Zwischenspeicher-String, einfach mal ausprobieren.

## then

Bestandteil der **if**-Zeile. Siehe [if](#).

## toggle / tO

<b>toggle relayx</b>	x: 1...16
<b>toggle Variable.Bit</b>	Bit: 0...31
<b>toggle Adresse, Bit</b>	Bit: 0...31

```
toggle relay3
```

Toggelt das Relais 3. War Relais 3 eingeschaltet, wird es ausgeschaltet. War es ausgeschaltet, wird es eingeschaltet. Programm und Direktmodus.

## touch

fragt den Touchscreen ab.

### touch X, Y

touch wartet auf eine Berührung des Touchscreens und übergibt die X-Koordinate in die 1. Variable sowie die Y-Koordinate in die 2. Variable. X und Y dürfen jede numerische Variable auch Array-Elemente sein. Innerhalb von Interrupt-Routinen wartet touch nicht.

## until / uN

Bestandteil der do / until Schleifenstruktur. siehe [do](#).

## usb

\*

### usb 5

usb 5 schaltet den USB als Device wie eine serielle Schnittstelle Nr. 5 ein. Der USB-Board-Anschluss sollte bereits per USB-Kabel mit dem PC verbunden sein, sonst wartet die Anweisung usb 5 max. 30 Sekunden auf die Herstellung der Verbindung. Auf dem PC (Windows10/11) wird der Basic-Computer als Serielles (CDC) Gerät erkannt und ein Universal-Windows-Treiber installiert. Das Gerät wird gestartet, aber nicht migriert – funktioniert jedoch problemlos. Für Windows7 ist ein zusätzlicher Treiber erforderlich, erhältlich auf Anfrage. Die Konsole wird nicht auf den USB umgeleitet.

Die Anweisung usb dient der Aktivierung des USB per Software, sie ist wirkungslos, wenn der USB-Anschluss bereits aktiv ist. (aktiver USB-Anschluss ist Standart nach der Erstinbetriebnahme - kann aber auch mit config dauerhaft ein- oder ausgeschaltet werden).

Ein einmal gestarter USB wird vom System überwacht und auch nach Unterbrechung der Verbindung sofort wieder reaktiviert, sobald die physische Verbindung wiederhergestellt ist. Der Zustand der Verbindung kann mit der Funktion sys(15) abgefragt werden. In eine unterbrochene Verbindung kann man nicht senden.

PC-seitig sollten evtl. vom Basic-Computer gesendete Daten auch abgeholt werden, print# oder put# warten darauf (notfalls bis in alle Ewigkeit) ☺.

\*

### usb 9

schaltet den USB als Host-Schnittstelle Nr. 9 ein. Man kann dann eine USB-Tastatur am USB anschliessen.

Voraussetzungen: 1.) Es ist eine kleine [Zusatzzplatine](#) zwischen USB-Pins und Tastatur geschaltet. 2.) Die Konsole arbeitet auf Schnittstelle 1.

usb 9 wartet auf den Anschluss der Tastatur. Diese Software-Aktivierung gilt nur bis zum nächsten Reset. Dauerhaft bitte mit config einschalten.

Die USB-Tastatur ahmt die PC-Eingabe via TeraTerm weitgehend nach. Mit input #9 oder get #9 holt man die eingetippten Zeichen ab.

usb funktioniert nicht in Interrupt-Routinen.

## user

### user Programm-Nummer

ruft ein Maschinensprache-Programm im ROM auf. Das Programm wurde zuvor mit receive "m" ins ROM geflasht. Die Programme stehen an

bestimmten festen Adressen: 0x000C0000 für Programm-Nr. 1, 0x000C4000 für Programm-Nr. 2, 0x000C8000 für Programm-Nr. 3 und 0x000CC000 für Programm-Nr. 4 alle im ROM-Sector 10). Steht an der aufgerufenen Adresse kein Programm schmiert das Basic ab. Ebenso wenn das dort befindliche Programm nicht ordentlich zurückkehrt oder unerlaubte Sachen macht 😊.

wait

wait Millisekunden

wait buf

`wait diginp Eingangs-Nummer` oder `wait diginp.Bit-Nummer`

wait Millisekunden

**ex.:**  
10 wait 1000

Wartet 1 Sekunde.

Das Programm wartet die angegebene Zeit in Millisekunden. Wertebereich 1...2147483647. Die Zeit ist Timer-gesteuert daher ziemlich exakt. (Abweichung nur durch die Ausführungszeit der Anweisung selbst). Innerhalb von Interrupt-Routinen hat `wait` keine Funktion. Achtung: `wait` setzt den Millisekunden Timer auf Null.

\*

`wait buf` wartet auf ein Zeichen im Empfangspuffer der Konsole:

**ex.:**

ersetzt: 10 do : until buf > 0

`wait buf` nur einsetzen, wenn das empfangene Zeichen nicht benötigt wird, also *allein* auf einen Tastendruck gewartet wird. (Nur `inkey` kann das Zeichen trotzdem holen).

\*

`wait diginp` dient zum Entprellen von Tastern, die an die digitalen Eingänge 1...12 (Bit-Nummer 0...11) angeschlossen sind. Bitte beachten: die Bit-Nummern der Systemvariablen `diginp` sind 1 kleiner als die Nummern der Digitalen Eingänge. `wait diginp` wartet, bis der Taster betätigt und wieder losgelassen wird:

```
ex.:
10 reset relay 2
20 do
30   wait diginpl
40   toggle relay 2
50 until
```

Startbedingung: Relay 2 aus  
ahmt einen Eltako nach:  
Taster 1 kurz betätigen  
abwechselnd ein und aus

wait digip wurde nur hinzugefügt, weil der Autor schreibfaul ist:

30 wait diginp1 macht exakt dies:  
30 while diginp 0=0 : wait20 : wend : while diginp 0=1 : wait 20 : wend

Innerhalb von Interrupt-Routinen hat `wait` diginn keine Funktion.

## wdog

wdog **timeout**                      timeout: 1...32 Sekunden

Watchdog, überwacht das Programm auf Absturz. Eine Art Totmann-Schalter für den Basic-Computer. Spätestens kurz vor Ablauf der **timeout**-Zeit muss der Watchdog-Timer (WDT) zurückgesetzt werden. Das erledigt jede Basic-Anweisung direkt nachdem sie aufgerufen wurde. Stürzt der µC während der Ausführung einer Anweisung ab, wird nach Ablauf der **timeout**-Zeit ein Reset gestartet, genauso, als ob der Reset-Taster gedrückt wurde. Der µC könnte sein Autostart-Programm neu laden und seinen Job fortsetzen.

**ex. :**  
10000 wdog 20

Die Anweisung **wdog** startet den Watchdog, stoppen lässt er sich dann nicht mehr (nur durch Druck auf den Reset-Taster). **timeout** sollte nach der langsamsten Basic-Anweisung im Programm bemessen werden (z.B. **gps** benötigt mindestens 2 Sekunden). Wartende Anweisungen (**get**, **input**, **server**, **wait** usw.) setzen den Watchdog regelmäßig zurück. Im Kommando-Modus macht **wdog** nicht allzuviel Sinn, ist aber möglich.

Der Watchdog schützt nicht vor Programmierfehlern und Programmen, die nicht mehr reagieren, weil sie in irgendeiner Schleife festhängen. Solange Anweisungen ausgeführt werden, ist alles in Ordnung.

Der WDT ist nicht sehr präzise – er wird von einem RC-Generator angetrieben.

## while wend

**while Bedingung : Anweisungen : wend**

Aus Kompatibilitätsgründen wurde die **while / wend** Schleife hinzugefügt. Die Anweisungen zwischen **while** und **wend** werden wiederholt ausgeführt, solange die Bedingung wahr (**<>0**) ist.

**ex. :**  
100 let A = 0  
1000 while diginp.0=0  
1010 inc A  
1020 wend  
1030 print A

Mehrere (max. 20) **while**-Schleifen können ineinander verschachtelt werden. **while** wird gerne verwendet, wenn die Einhaltung einer Bedingung überwacht werden muss (z.B. ob ein Eingang einen bestimmten Wert behält). Eine ewige Schleife erhält man z.B. mit:        10 while true

## write

Für das platzsparende Speichern von Zahlen in sequentiellen Dateien gibt es **write#**. Gelesen werden die Zahlen mit **read#**. Beschreibung siehe [fn Open\(\)](#).

# Variablen

## numerische Variablen

**A, B, C, D, E, F, ...Z**

pahlbasic kennt die 26 vordefinierten Variablen **A, B, C, ...Z**, die numerische Werte (Fließkomma, 64Bit) aufnehmen können. Diese Variablen müssen vor ihrer Verwendung weder deklariert noch initialisiert werden. Ihr Startwert ist **0**.

Innerhalb von Prozeduren und Funktionen gibt es einen zusätzlichen halben Satz lokaler Variablen (**A ...J**), die unabhängig von den globalen Variablen sind. Werte erhalten alle Variablen mit der Anweisung **let**:

```
ex.:
10 let B = 23
```

Zusätzlich gibt es maximal 24 weitere Variablen, die individuelle Namen haben dürfen. Gekennzeichnet werden diese Variablen durch das vorangestellte Zeichen **\$**. Der **Name** darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen, max. 16 Zeichen und muss mit einem Grossbuchstaben beginnen:

```
ex.:
10 let $Num_Zeiger = 23.5
20 let $X2 = 115
```

Diese Variablen werden automatisch angelegt, wenn sie (z.B. mit **get**, **let**, **sub**, **function**, **input**, **read**) einen Wert zugewiesen bekommen. Sie sind **global**. Als **for/next**-Schleifen-Zähler funktionieren diese Variablen nicht. Treffen Namensvariablen und logische Operatoren (**and**, **or**, **xor**) aufeinander, Leerzeichen zur Trennung einfügen. Eine Liste dieser Variablen zeigt man mit: **dump V** an. (Das Zeichen **\$** findet man jetzt auch auf Funktionstaste F9)

**Gleich ein ganzes Feld von Variablen erhält man mit den Arrays, deren einzelne Elemente man über einen Index (oder 2 Indizes) ansprechen kann:**

## Arrays

es gibt zehn ein-dimensionale Arrays    **A( ) ... J( )**   mit jeweils 100 Elementen **\***  
(Index: **0...99**) z.B.: **F(15)**

```
ex.:
10 let F(15) = 99
```

Der Index darf natürlich in der Klammer berechnet werden.

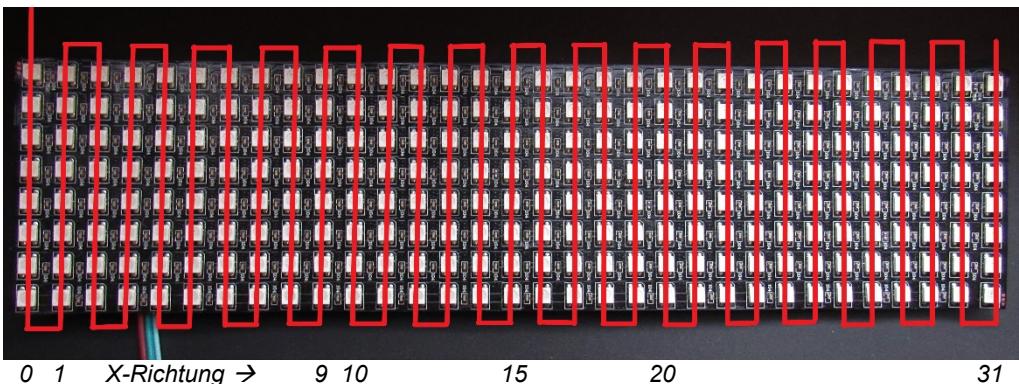
```
ex.:
10 print "x", : print "sqr(x)"                                Quadratwurzel-Tabelle
20 for P=0 to 9 : let A(P) = sqr(P+1) : next P
30 for P=0 to 9 : print P+1, A(P) : next P
In diesem Beispiel ist A()
ein Array und P eine einfache
Variable
```

Zusätzlich gibt es die besonderen Arrays **L()**, **P()**, **S()**, **T()** und **Z()** mit speziellen Eigenschaften:

**Das neue Array L()** enthält die Farbwerte der Neopixel LEDs. Es hat 256 Elemente zu 32Bit. Dieses Array kann 1-dimensional oder 2-dimensional angesprochen werden.

Nur eine Dimension benötigt ein LED-Streifen, der Index läuft von 0...255 entspricht LED 1...256. Die Farbwerte enthalten die Werte für den Rot-, Grün- und Blau-Anteil als 32-Bit-Zahl. Siehe Programmbeispiel [Function\\_RGB\\_Test](#).

Zwei Dimensionen werden für LED-Panel benötigt. Zurzeit werden LED-Felder mit 256 LEDs in der Anordnung von 8x32 LEDs unterstützt. Die Indizes entsprechen den Koordinaten der LEDs  $L(x, y)$ . In X-Richtung 0...31 und in Y-Richtung 0...7. LED 1 ist die LED mit den Koordinaten 0,0. Diese LED am Koordinaten Ursprung ist entweder links oben oder links unten – konfigurierbar mit [config led](#). Die Panel-LEDs müssen meanderförmig verdrahtet sein. In einer späteren Version wird auch Schrift unterstützt.



Das Array **P()** ist die Farbpalette und hat z.Z. 64 Elemente (Index 0 ...63). Die Elemente 0 ...22 sind vom System vorbelegt. Der **Index** ist die Farbnummer. Siehe Anhang [Farbpalette](#). Siehe [draw](#)

**S()** ist eine Zugriffsmöglichkeit auf den Soundspeicher. Der Soundspeicher enthält die Stützstellen/Samples für eine Tonschwingung. Der Soundspeicher belegt die Indizes 0 ...1023. Man greift darauf zu, wie auf die „regulären“ Arrays: z.B.: let **S(444) = 128**

Für die Soundausgabe müssen die Array-Werte im Bereich von + 2047 bis -2048 (12-Bit) liegen, andernfalls produziert der DA-Wandler sehr unangenehme Verzerrungen. Wird die Soundausgabe nicht genutzt, kann man es als „fast normalen Zahlenspeicher“ missbrauchen. (aber nur 16-Bit Integer +32767 ...-32768)

**T()** ist ein **Text**-Array, die Elemente sind **Strings** - [siehe unter Strings](#).

**Z()** ist wie die Arrays **A()** ...**J()**. Mit zwei wichtigen Unterschieden:

Beim F405/7 ist das Array **Z()** im batterie-gepufferten RAM untergebracht. Dort passen 500 Fließkommawerte 1-dimensional: Index 0 ...499 oder 10x50 Werte also 2-dimensional: Index A 0 ...9 und Index B 0 ...49 hinein.

Es behält seine Werte auch nach dem Ausschalten der Versorgungsspannung. Zum Löschen gibt es die [clear-Anweisung](#) (**clear Z**)

let **Z(499) = 234** 1-dimensional      print **Z(9, 49)** 2-dimensional

\*) Die Arrays **A-J** sind im RAM ein durchgehendes Zahlenfeld von  $100 \times 10 = 1000$  Zahlen (8KBytes). Die Einteilung in 10 Arrays mit je 100 Elementen ist willkürlich.

Man kann sie alternativ auch als ein einziges 1-dimensionales oder 2-dimensionales Array **K()** ansprechen.

1.) eindimensional:

**K(x)**    x=0...999

erstes Element:

**K(000)** entspricht **A(0)**

**K(100)** entspricht **B(0)**

**K(200)** entspricht **C(0)**

...

...

...

**K(900)** entspricht **J(0)**

... letztes Element:

... **K(099)** entspricht **A(99)**

... **K(199)** entspricht **B(99)**

... **K(299)** entspricht **C(99)**

...

...

...

... **K(999)** entspricht **J(99)**

2.) zweidimensional:

**K(x, y)**            x=0 ...9 (wie A ...J)        y=0 ...99

**K(0, 0)** entspricht **A(0)**            ... **K(0, 99)** entspricht **A(99)**

**K(1, 0)** entspricht **B(0)**            ... **K(1, 99)** entspricht **B(99)**

**K(2, 0)** entspricht **C(0)**            ... **K(2, 99)** entspricht **C(99)**

...

...

...

**K(9, 0)** entspricht **J(0)**

...

...

...

... **K(9, 99)** entspricht **J(99)**

Die verschiedenen Darstellungen *derselben* Elemente können gemischt eingesetzt werden – also:

let **B(14)** = 234            print **K(1, 14)**            print **K(114)**

sprechen alle dasselbe Element an. Das Array **K()** ist also kein zusätzliches Array! Es erleichtert lediglich den sequentiellen Zugriff auf sämtliche Array-Elemente.

## Zeichenketten Variablen

**A\$, B\$, C\$, D\$, E\$, F\$, G\$, H\$**

können jeweils 254 Zeichen aufnehmen. **H\$** hat für einige Instruktionen eine besondere Aufgabe. **A\$** und **B\$** sind innerhalb von Prozeduren und Funktionen lokal.

```
ex.:
10 let A$ = "Garage" : print A$
20 print (A$)
```

Die Zeichenkette "Garage" wird im String **A\$** gespeichert und dann ausgedruckt  
drückt den ASCII-Wert des 1.Zeichens aus – siehe Klammer

## Text Array

es gibt *ein Text-Array*:

**T()**

Dient zur Aufnahme von längerem **Text** (16 KByte), es besteht aus 128 Strings (0...127) mit je 126 Bytes Länge. (zuzügl. Nullbyte)

```
ex.:
5 let T(100) = "Dies ist ein Text"

10 let N = 0
20 do : print T(N) : inc N : until (T(N))=0
```

drückt das Text-Array auf dem Terminal aus (bis zum ersten Leer-String)

## Systemvariablen

ad1

ADC1

Wert des 1. Analog-Digital-Wandlers. 0 = 0Volt, 4095 = + 3,3 Volt.  
Nur lesen. Auflösung 12bit.

```
ex. :  
10 print ad1/4095*3.3; " Volt"
```

ad2

ADC2

Wert des 2. Analog-Digital-Wandlers. 0 = 0Volt, 4095 = +3,3 Volt.  
Nur lesen. Auflösung 12bit.

```
ex. :  
10 let A = ad2/4095*3.3  
20 if ad2<512 gosub 1000
```

ad3

ADC3

Wert des 3. Analog-Digital-Wandlers. 0 = 0Volt, 4095 = +3,3 Volt.  
Nur lesen. Auflösung 12bit.

```
ex. :  
10 print ad3/4095*3.3; " Volt"  
20 if ad3<512 gosub 1000
```

ad4

wie ad1

attack

Attack-Zeit: erste der 4 Phasen des [ADSR-Hüllkurven-Generators](#). Der eingebaute Hüllkurvengenerator arbeitet mit der Anweisung [play](#) und der MIDI-Schnittstelle zusammen. Siehe auch: [decay](#), [sustain](#), [release](#). Es soll eine Art einfacher [SID-Chip](#) in Software nachgebildet werden.

```
ex. :  
10 let attack = 20
```

Anstiegszeit in Prozent der Gesamtnotenzeit. Wertebereich 2...98%. 98% heisst, der Ton schwilkt bis zum Ende der Note an. Ergibt die Summe von [attack](#) + [decay](#) mehr als 100% wird [decay](#) entsprechend korrigiert.

buf

Buffer. Anzahl der Zeichen im **Konsolen**-Empfangspuffer. (Serielle Schnittstelle 1 oder USB). Die Zeichen werden nicht abgeholt. Lesen und Setzen. (Setzen - z.B. auf Null: [let buf = 0](#))

```
ex. :  
10 if buf>0 gosub 300  
20 on com start SerInt
```

Die Empfangspuffer der einzelnen Schnittstellen kann man abfragen:

`sys(21)`: serielle Schnittstelle 1  
`sys(22)`: serielle Schnittstelle 2 (MIDI)  
`sys(23)`: serielle Schnittstelle 3 (GPS)  
`sys(24)`: serielle Schnittstelle 4  
`sys(25)`: USB abfragen.

löschen kann man die Empfangspuffer mit `clear buf x` (`x` = 0...5, 9)

## clock / clo

Uhrzeit

Uhrzeit der Hardware-Uhr als Zeichenkette als HH:MM:SS. HH = Stunden, MM = Minuten, SS = Sekunden.

```
let clock = "14:35:20"          stellt die Hardware-Uhr (RTC).
```

**ex.:**  
10 let A\$ = clock  
20 print clock  
30 if (clock="06:00:00") then impuls relay1,1000

`clock` ist eigentlich eine **Stringfunktion**. (wie auch [date](#))

## color / col

Farbnummer für die Ausgabe mit `text`. `0` = schwarz, `9` = Weiss. Ist die Farbpalette ausgeschaltet, den 16Bit-Farbwert angeben.

```
10 let color = 9 : text 100, 100, "Hallo Welt"
```

## dac

12-Bit Analogausgang, gibt eine Spannung in 4096 Schritten von 0V bis 3,3V aus. 0=0V, 4095=3.3V. Ist der interne DAC (PortA Pin 4+5) von der Soundausgabe belegt, wird mit Fehlermeldung abgebrochen.

**ex.:**  
10 let dac = 1025

## dataptr / daT

Datenzeiger für die [read](#) Anweisung. Prozeduren und Funktionen haben einen eigenen lokalen Datenzeiger. Will man diesen an das Hauptprogramm übergeben, kann man den Datenzeiger in der Funktion / Prozedur in eine Variable lesen und im Hauptprogramm zurückschreiben. Das wirkt dann wie eine [restore](#) Anweisung.

**ex.:**  
10000 sub IrgendeineProzedur  
10010 read A, B, C : print A, B, C  
10020 let \$D\_Zeiger = dataptr in der Prozedur  
10030 data 201, 405, 28.5, 4711, 815, 12  
10040 end  
  
3100 let dataptr = \$D\_Zeiger im Hauptprogramm  
3110 read A, B, C : print A, B, C liest da weiter, wo die  
 Prozedur aufgehört hat

## date

Datum der Hardwareuhr. `date` ist ein **String** mit dem Format:

“Wochentag, TT.MM.JJJJ“ (de) oder “weekday, MM.DD.YYYY“ (us)

Beim **Setzen** muss der Wochentag nicht angegeben werden, aber bitte immer im deutschen Format TT.MM.JJJJ :

```
let date = "20.12.2021"      setzt das Datum der Hardware-Uhr (RTC)  
print date                  druckt das Datum: Fri, 12.20.2021  
                            (hier US-Format)
```

## day

Tag

## decay

Decay-Zeit: Zweite der 4 Phasen der ADSR-Hüllkurve. Siehe auch: [attack](#), [sustain](#), [release](#). Es soll eine Art einfacher [SID-Chip](#) in Software nachgebildet werden.

```
ex.:  
10 let decay = 60
```

Abkling-Zeit in Prozent der Gesamtnotenzeit. Wertebereich für `decay`: 2...98. Ergibt die Summe von `attack` + `decay` mehr als 100% wird eine Fehlermeldung ausgegeben.

## diginp / diG

Digitaleingänge 1 ...12

`diginp.x`                    `x` = 0...11

Die Systemvariable `diginp` ist ein 12-Bit Wert, der den Zustand der 12 digitalen Eingänge widerspiegelt. `diginp` entspricht dem Port E (GPIOE\_IDR). Jedes Bit entspricht einem Eingang (Bits 0...11). Auf Bits greift man mit dem Dezimalpunkt zu.

```
ex.:  
10 if diginp.0 then return          Reagiert auf Zustand  
                                    von Eingang 1  
  
20 for N=0 to 11 : print diginp.N : next N  
                                Druckt den Zustand  
                                aller Eingänge aus.
```

Man kann in `diginp` auch schreiben, dann werden für die 1ser-Bits die Pullup-Widerstände und für die 0er-Bits die Pulldown-Widerstände eingeschaltet. Nach einen Reset sind alles Pulldowns. Siehe auch [config](#) für eine dauerhafte Einstellung.

```
let diginp.2 = 1      schaltet Pullup auf Eingang 3 ein  
let diginp = 4095    schaltet alle 12 Pullups ein  
let diginp = 0        wieder alles Pulldowns
```

Die Eingänge 1...5 (`diginp` Bits 0...4) können `on`-Ereignisse auslösen, siehe Anweisung [on](#)

dow

Day of week

Wochentag der Systemuhr. Werte: 1...7 . Montag = 1.

erl

Fehlerzeile

Nummer der Programmzeile, in der der letzte Fehler aufgetreten ist.  
Ist kein Fehler aufgetreten, wird die aktuelle Zeilen-Nummer ausgegeben.

err

Fehlernummer

Nummer des zuletzt aufgetretenen Fehlers. err = 0 heißtt: kein Fehler aufgetreten. Lesen und setzen möglich – setzen erzeugt künstlich einen Fehler:

2000 let err = 31 (soft break=Zwangsabbruch).

```
ex.:
10 on err gosub 1000
20 input "Eingabe : ", A : let A = 1/A
30 'testweise 0 eingeben.
....
1000 rem Fehlerbehandlung
1010 resume 20           Eingabe wiederholen.
```

fnvar, result / fN, rE

fnvar ist das Argument und result das Resultat von User-Funktionen. Zwei Seiten einer Medaille – funktionieren nur innerhalb des Funktionskörpers. Siehe auch: [user-defined-Functions](#).

font

Zeichensatz. Es gibt 8 [Fonts](#): font 0...font 7. (mit dump 2 abfragen)

```
let font = 4           font 4 ist Arial bold 21x24 Punkte
text 10, 10, "pahlbasic"  Standardfont ist font 0. (Tahoma 10x12)
```

Fonts Nr. 8 bis 15 sind eine Wiederholung der Fonts 0...7. Es wird aber um 90° gedreht von unten nach oben geschrieben. Fonts 16 bis 23 schreiben kolumnenartig von oben nach unten.

hour

Stunden

Stunden der Hardwareuhr. Nur 24-Stundenmodus. Nur Lesen.

intrpt

Nummer des aktuellen Interrupts (1...24). Nur innerhalb von Interrupt-Service-Routinen abfragbar. Nur Lesen.

Siehe auch: [sys\(95\)](#), [reset intrpt](#), [set intrpt](#)

minute / miN

Minuten

Minuten der Hardwareuhr. Nur Lesen.

milli

Milli-Sekunden-Zähler. Lesen und setzen. Zählbereich: 31 Bit = 596 Stunden. **milli** wird nicht durch Interrupts erzeugt. (freilaufender Timer2). **milli** wird von Anweisungen wie wait, impuls, play, morse etc. zurückgesetzt.

month / moN

Monat.

pgm

Nummer der im Speicher befindlichen Programm-Datei. Siehe auch dir. 0 = Programm-Nr. 0 oder wurde gar nicht geladen, sondern wird gerade neu im RAM erstellt. Nur Lesen. **Wird abgelöst von sys(200)**.

```
print pgm
```

**ex.:**  
10 load pgm +1

nächstes Programm nachladen

pwm1, pwm2, pwm3, pwm4

Pulsweitenmodulator 1, 2 und 3 (auf dem F405RG, F407ZG und dem F411/F401 auch **pwm4**)

Momentaner DutyCyclewert der Pulsweitenmodulatorkanäle 1, 2, 3 und 4. Die Frequenz und die Startwerte nach einem Reset sind jetzt konfigurierbar siehe config. Die 'Auflösung' der PWM ist abhängig von der Frequenz. Sie kann mit sys(14) abgefragt werden. Setzen eines neuen Wertes geschieht asynchron.

z.B.: **pwm1** auf 50% setzen:

**ex.:**  
10 let pwm1 = sys(14) \* 50 / 100

Die Anweisung **servo** wurde entfernt. Stattdessen setzt man nun die Frequenz der PWMs auf 50Hz (20ms) für analoge Servos und sorgt für eine Impulsweite von 1..2ms – 1ms entspricht dem rechten Anschlag des Servos, 2ms dem linken.

Tatsächlich weichen die Werte bei unterschiedlichen Fabrikaten stark ab. Getestet wurden verschiedene Servos, bei denen die Impuls-Werte von 0,5ms bis 2,5ms reichen.

**ex.:**  
10 config pwm1=50, 1500 (50Hz, Startwert 1500)  
20 let pwm1 = 8000 (ca. 2,5ms)  
30 wait 1000  
40 let pwm1 = 1500 (ca. 0,5ms)

Digitale Servos vertragen in der Regel auch 300Hz (3,3ms) und PWM-Werte von 7000...45000. Auch hier muss man etwas experimentieren, um die optimalen Werte zu ermitteln.

Etwas Mathe gefällig? **print sys(14)** liefert den maximalen Zählerstand der PWM. Er entspricht der Impuls-Gesamtlänge.

Bei 50Hz liefert **print sys(14)** den Wert 63529. Dies entspricht also 20ms. 1ms entspricht  $63529 / 20 = 3176$ , 2ms demnach 6352.

Ideale Servos sollten mit diesen beiden Werten den rechten und linken Anschlag erreichen. (und dabei 180° abfahren)

```
let G = 20                                     G = Winkel in Grad
let pwm1 = G*3176/180+3176
```

Die PWM-Ausgänge sind ohne Eingriff sofort nach dem Reset aktiv.

## relay / rel

Zustand der 'Relais'. Die Systemvariable `relay` enthält die Zustände der 16 'Relay'-Ausgänge als Bits 0...15. Entspricht dem Port D des Prozessors (Register: GPIOD\_ODR / GPIOD\_IDR)

```
print relay.1                         Zustand von: Relay Nr.2
let relay.12 = 1                       Alternativ zu: set relay13
```

Nach einem Reset ist `relay` = 0, d.h.: alle Portpins sind ausgeschaltet. Dies lässt sich mit der Anweisung `config` ändern.

Seitdem die Anweisung `blink` entfernt wurde, können die 'Relay'-Ausgänge für jede Art Steuerung/Impulserzeugung verwendet werden – kein Interrupt pfuscht mehr dazwischen. Der Name 'relay' ist nur noch historisch. ☺

Die digitalen Ausgänge sind ohne Eingriff sofort nach dem Reset aktiv.

## rand

Random Number Generator

Gibt eine 32-Bit Zufallszahl aus, vom Hardware-RNG geliefert (nur F405/7). Ein digitaler Würfel gefällig? `print int(rand/true*6)+1`

Für den F411/F401 siehe auch [seed](#).

## release / rL

Release-Zeit: 4. Phase des [ADSR-Hüllkurven-Verlaufs](#). Siehe auch: [attack](#), [decay](#), [sustain](#). Es soll eine Art einfacher [SID-Chip](#) in Software nachgebildet werden.

```
ex. :
10 let release = 30
```

Nachklingzeit in Prozent der Gesamtnotenzeit. Wertebereich von `release` 2...10000.

## second / seC

Sekunden

Sekunden der Hardwareuhr. Lesen.

## speed

Ausgabegeschwindigkeit der [morse-Anweisung](#). Anzahl Worte pro Sekunde. Maßstab ist das Wort „Paris“. Defaultwert nach jedem Reset ist 20.

```
ex. :
10 let speed = 40
```

stack

Der `gosub`- / `loop` -Stapel auf den pahlbasic die Rücksprung-  
adressen legt, hat einen Ebenen-Zähler: `stack`  
Lesen ist unkritisch, Schreiben kann zu Fehlermeldungen führen.

## sustain / suS

Halte-Level der Sustain-Phase (Nr.3) der ADSR-Hüllkurve. Siehe auch: [attack](#), [decay](#), [release](#).

**ex.:**

### Sustain-Level in Prozent. (0...100%)

tempo / teM

Geschwindigkeit, mit der Töne/Noten abgespielt werden.

**ex.:** 10 let tempo = 120 Notentempo = 120 Schläge/Minute

Wertebereich 15...360.

Bestimmt nur die Geschwindigkeit des Hüllkurvengenerators und damit wie lange eine Note dauert. Das Melodietempo bestimmt der User mit der on-milli-Interrupt-Frequenz seiner Melodie-Abspiel-routine.

time

## Timer der Systemuhr

**time** wird nach dem Einschalten des Basic-Computers jede Sekunde um 1 erhöht. Auflösung: 32 Bit. Setzen und lesen.

```
? time          Lesen  
let A = time    Lesen
```

**ex.:**

```
10 let time = 0 Setzen  
20 if time>A goto 200 Lesen
```

urgend

Dringlichkeit.

Noch in der Entwicklung.

volume / v<sub>0</sub>

Lautstärke, `volume` stellt die Lautstärke des abgespielten Sounds in Prozent ein. `volume` kann Werte von 0 ...100 annehmen. Wenn die DACs Verzerrungen erzeugen, `volume` verringern z.B. auf 50. Siehe [sound](#).

year

Jahr.

Im Gegensatz zu 'normalen' Variablen haben die meisten Systemvariablen keine Adresse im RAM sondern werden im Moment der Abfrage oder des Setzens generiert.

## Zugriff auf einzelne Bits

auf ein einzelnes Bit einer **numerischen** (System-)**Variablen** greift man mit dem Dezimalpunkt zu:

(System-)**Variable**.Bitnummer

```
ex.:
10 let A.15 = 1
20 print relay.1
30 for N=0 to 31 : print C.N : next N
40 if J.3=0 then print "Bit3 ist 0"
```

einfache numerische Variable  
Systemvariable  
Bit-Nr ist eine Variable  
Bit prüfen

Die entsprechende (System-)**Variable** wird *vorher* in eine **32-Bit-Wort-Zahl** (uint32) gewandelt. Der Wert hinter dem Punkt muss kleiner als **32** sein. (Bit **0...31**, die **Bits** zählen von rechts) und darf auch eine Rechenformel sein (am Sichersten in Klammern). Das Bit kann nur die Werte **0** oder **1** annehmen:

`let relay.2*N+1 = 1` die Bitnummer wird ausgerechnet, ist der Wert größer als 31 wird mit einer Fehlermeldung abgebrochen.

`print relay.(2*N+1)+X` Hier wird der Rechenausdruck nach der Bit-Nummernberechnung weiter fortgesetzt. Die Klammer begrenzt die Bit-Nummern-Berechnung, +X gehört nicht mehr dazu.

oder so:

`print(relay.2*N+1)+X`

Bit-Test:

`if relay.(2*N+1)>0 then ...` bei Bit-Tests oder -Vergleichen nur die Operatoren **=, <>, >, <** verwenden!

Bit-Vergleich:

`Bit=Bit, Bit<>Bit, Bit>Bit, Bit<Bit`

`if Variable1.(Rechenausdruck)=Variable2.(Rechenausdruck) then ...`

```
ex.:
if A.0<>B.0 then ...
print relay.(2*N+1)=diginp.(N+1)
```

siehe auch: [Bits und Bytes](#) und die neue Funktion: [fn Testbit\(Ausdruck, Bit-Nr.\)](#)

# Operatoren

## arithmetisch

keine Rechenregelbeachtung, bitte die gewünschte Reihenfolge mit Klammern erzeugen.

- + Addition
- Subtraktion
- \* Multiplikation
- / Division
- ^ Potenz z.B. das Ziehen der 12.Wurzel: let A = 2 ^ (1 / 12)
- add** Integer-Addition, die Summanden sind Integerzahlen (werden abgerundet)
- div** Integer Division. **Divident**, **Divisor** und **Ergebnis** sind Integerzahlen – Fließkommazahlen werden abgerundet:  
print 7 div 2 Ergebnis: 3
- exp** Integer-Exponent – wenn 2 hoch 3 wirklich 8 sein soll ☺  
print 3 exp 3 Ergebnis: 27
- mod** liefert den Rest einer Division ganzer Zahlen.
- mul** multipliziert 2 Integerzahlen. Das Ergebnis ist ebenfalls Integer.
- sbt** Integer-Subtraktion
- +=** beliebte Abkürzung aus der Programmiersprache C. Addiert einen konstanten **Betrag** zur (System-)**Variablen** und setzt ihren neuen Wert in den Rechenausdruck ein:

```
10 if A += 5 > 10 then...
```

ersetzt diese beiden Anweisungen:

```
10 let A = A + 5
20 if A > 10 then...
```

Die folgenden Operatoren funktionieren equivalent zu **+=**:

**-=**, **\*=**, **/=**, **&=**, **|=**, **^=**, **<<=**, **>>=**

**&=** entspricht: **and=**,    **|=** entspricht: **or=**,    **^=** entspricht: **xor=**  
(**and=** und **or=** sowie **xor=** gibt es aber nicht)

Die Syntax ist für alle Abkürzungs-Operatoren gleich:

(System-)**Variable Operator Zahl**

Eine Klammer ist nicht unbedingt nötig. Hält man sich *nicht* an obige Syntax, werden die Operator-Zeichen getrennt abgearbeitet und ergeben völlig andere Ergebnisse. Im Gegensatz zur Programmier-

sprache C, können diese Abkürzungen nicht für sich alleine stehen, sondern werden in Anweisungen wie `print`, `if`, `until` oder `while` und in Rechenausdrücken mit `let` eingesetzt:

statt: `let A = A+5` geht auch: `let A+=5`

- <<** bitweises links-schieben (Das Ergebnis ist ein Longwordwert (32-Bit). Der rechte Operand (der angibt um wieviel Bits verschoben wird) darf nicht grösser als 32 sein). Von rechts werden Nullen nachgeschoben, herausgeschobene Bits fallen weg:

```
10 let A = true
20 print bin$(A, 32), bin$(A << 3, 32)
```

- >>** bitweises rechts-schieben (sonst wie <<). Von links werden Nullen nachgeschoben, herausgeschobene Bits fallen weg:

```
10 let A = true
20 print bin$(A, 32), bin$(A >> 3, 32)
```

- @** Pointer, liefert die physische Adresse z.B. einer Programmzeile:

<code>let A = @1000</code>	<code>let A = @"func:Labelname"</code>
<code>let A = @"proc:Labelname"</code>	<code>let A = @"sub:Labelname"</code>

oder einer **Variablen / Arrayelement / Stringvariablen / Zeichen**:

<code>print @A</code>	<code>print @Z(295)</code>
<code>print @B\$</code>	<code>print @T(99)</code>
<code>print @B\$(3)</code>	<code>print @T(99, 3)</code>
<code>print @\$NamenVariable</code>	

### Zeilenadresse

dient z.B. der Beschleunigung der Programm-Ausführung bei `goto` und `gosub`. Wird eine Routine (Unterprogramm) häufig aufgerufen, bringt der Aufruf der Routine mit ihrer Adresse einen Geschwindigkeitsvorteil:

<b>ex. :</b>	
<code>10 rem Adress-Speicherung</code>	
<code>20 let A = @1000</code>	ermittelt die Adresse von Zeile 1000 und speichert sie in A
<code>100 gosub -&gt;A</code>	hier springt gosub zur Adresse, die in A gespeichert ist, (ohne suchen)
<code>....</code>	
<code>1000 rem Messwertausgabe</code>	hier die Zeile, deren Adresse
<code>1010 let C = 2*B/100 : print C</code>	gespeichert wurde
<code>1020 return</code>	

Wie müsste das Programm wohl geändert werden, wenn auf Zeilennummern verzichtet werden soll?

on-Interrupts, User-Funktionen und -Prozeduren arbeiten von Haus aus mit Adressen, benötigen keine extra "Beschleunigung".

- Pointer liefert / schreibt den Inhalt einer Adresse. Gegenstück zu `@`. Wird in Anweisungen wie `break`, `goto`, `gosub`, `restore` als Zeilenadresse interpretiert. Als Pointer können nur die Variablen `A ...Z` dienen:

```
10 let A = @B(0) : let ->A = 44.5          A ist pointer auf B(0)
20 print ->A, B(0)                          B(0) enthält: 44.5

30 let A = @100                            A ist pointer auf Zeile 100
40 let A$ = ->A                           A$ enthält Zeile 100
50 goto ->A                             springt zur Adresse in A (Zeile 100)
```

Ist die Adresse dem Basic nicht bekannt, kann man einen Standart-Pointer konfigurieren. Das Rechnen mit Pointern ist aber ein ganz eigenes Thema. Als Anregung: `->(A+8)`.

Natürlich sind die Pointer in pahlbasic keine echten Pointer, aber sie funktionieren sehr ähnlich.

## logisch

die Operanden werden *vor* der Operation in **32-Bit-Werte ohne Vorzeichen** gewandelt. Achtung bei negativen Werten! Die Werte werden nur auf 32-Bit-Format zurechtgestutzt. Negative Zahlen werden bekanntlich durch ihr 2-er Komplement dargestellt (das höchste Bit repräsentiert das Minuszeichen):

-9 wird zu: `not(9)+1` binär: %1111111111111111111111110111

Zurückverwandelt in eine Fliesskommazahl ergibt dies aber: 4294967287, weil das Basic von einer 32-Bitzahl *ohne* Vorzeichen ausgeht. Richtig ist:

`print -1 * not(%1111111111111111111111110111) -1`

(in anderen Basic-Dialekt werden Integerzahlen verwendet, da tritt dieses Problem nicht auf)

!

bitweises `not` (Nicht), das Ergebnis ist ein **32 Bit-Wort-Wert**.

```
ex.:
10 print !5
20 let A.1 = !A.1
30 let A = 255 and (!5)
```

Wenn 2 **Operatoren** (`and` und `!`) aufeinander treffen, bitte Klammern setzen, sonst wirkt nur der erste Operator (`and`) ähnlich wie beim [Minus-Zeichen](#), das vom Prinzip auch ein unärer Operator ist. Man kann auch die Funktion `not()` benutzen (Rückwärtskompatibilität mit Version 1.03)

### and, &&

bitweises `and` (Und), das Ergebnis ist ein **32 Bit-Word-Wert**.

```
? 3 && 255
```

```
ex.:
10 print 3 and 255
20 let A = B and (5 or 8)
```

### or, |

bitweises `or` (Oder), das Ergebnis ist ein **32 Bit-Word-Wert**.

```
? 3 | 8
```

```
ex.:
10 print 3 or 8
20 let A = B or (5 and 255)
```

### xor,

bitweises `xor` (Exklusiv Oder), das Ergebnis ist ein **32 Bit-Word-Wert**.

Strenggenommen sind diese logischen Operatoren auch arithmetisch, weil Basic keine booleschen Werte kennt.

## Vergleich

>	grösser	print 6>5
<	Kleiner	print 5<6
>=	grösser gleich	print 6>=A
<=	kleiner gleich	print 6<=A
=	gleich	print A=B

nicht zu verwechseln mit dem Zuweisungszeichen = in der Anweisung let.

<>	ungleich	print A<>B
----	----------	------------

In anderen Basic-Dialekt sind Vergleichsoperatoren rangniedrig – nicht so bei pahlbasic.  
Alle Operatoren sind gleichwertig, die Reihenfolge bestimmt man mit Klammern selbst.

**ex.:**  
10 let A = (9\*3)>(14+7)

## Funktionen

`abs()`

liefert den Absolut-Wert eines Real-Wertes zurück.

```
print abs(-1.6)          Ergebnis: 1.6
```

`atn()`

berechnet den Arkustangens eines Radian-Wertes.

```
? atn(.3)          Ergebnis: 0.29145
```

`calc()`

`calc(Stringvariable)`

`calc()` liefert den **numerischen** Wert, den der *Inhalt* von **Stringvariable** repräsentiert. Der *Inhalt* von **Stringvariable** muss ein berechenbarer Ausdruck sein. Ist der Inhalt von **Stringvariable** nicht **numerisch**, wird mit Fehlermeldung abgebrochen. Vergleiche auch mit `val()`. Siehe auch: `num()`.

```
ex.:
10 let A$ = #125.45#
20 let A = calc(A$)          Zahlen, innerhalb von #-Zeichen,
                             dann
                             verhält calc() sich ähnlich wie val()

30 let B$ = #sin(pi/8)#
40 print calc(B$)          Rechenausdrücke innerhalb von
                           #-Zeichen, dann
                           berechnet calc() den Ausdruck
```

`calc()` kann auch mit den Elementen des **Text-Arrays** umgehen:

```
10 let T(0) = #sin(pi/4)*8#      T(0) ist Text Nr. 0 imTextarray
20 print calc(T(0))           Ergebnis : 5.65685
```

`calc()` kann selbst auch wieder in mathematischen Ausdrücken benutzt werden:

```
let A = 3*calc(A$)+2
```

`calc()` funktioniert auch mit Teilstrings.

```
10 let A$ = #3+4+5+6#
20 let B = calc(A$(3, 3))      Teilstring: 4+5
30 print B                      Ergebnis: 9
```

## cos()

berechnet den Cosinus eines Radiant-Wertes

```
? cos (.234)
```

**ex.:**

```
10 print cos(pi/4+B)
```

## epoch()

rechnet einen beliebigen \*) Zeitpunkt aus Datum & Uhrzeit in UNIX-Zeit (Sekunden seit dem 1.1.1970) um. Datum & Uhrzeit müssen Inhalt eines Strings sein. Der String muss so aufgebaut sein: "Datum,Uhrzeit" ohne Leerzeichen, genau 19 Zeichen lang:

Datum: TT.MM.JJJJ      Uhrzeit: HH:MM:SS

```
10 let A$ = "14.08.2022,11:55:10"  
20 print epoch(A$)
```

Unix-Zeit von jetzt \*\*):

```
10 let A$ = date(0) + "," + clock  
20 print epoch(A$(5,19))
```

blendet Wochentag aus

**epoch()** dient zum einfachen Vergleich (älter/jünger) von Datums- und Zeitangaben insbesondere bei Dateien. Gegenstück zu **epoch()** ist **epoch\$()**.

Auch **dow\$()** kann mit dem **epoch()**-Wert etwas anfangen (aber nur 01.01.2001 ... 31.12.2037).

\*) Datum sollte im Bereich von +/- 68 Jahre zum 1.1.1970 liegen. Daher röhrt das "2038-Problem" von 32Bit-Systemen.

\*\*) Geht jetzt auch als Einzeler (siehe Stringoperator **mid**):

```
10 print epoch(mid5,10,date(0) + "," + clock)
```

**date(0)** ist das deutsche Datum, unabhängig vom konfigurierten Format (**date(1)** ist das us-englische).

## fn1() ...fn5()

User-Funktionen werden vom User definiert. User-Funktionen vom Typ I verhalten sich wie die einfachen numerischen Funktionen (wie z.B.: `sin()` ). Siehe [User-Funktionen1](#).

**ex.:**

```
10 print fn1(22.5)
```

## fn Name(), fr Name()

User-Funktionen werden vom User definiert. [User-Funktionen vom Typ II](#) können bis zu 16 Argumente verarbeiten. Die Namen der eingebauten fn Funktionen wie z.B. `fn Attrib()` sind "Tabu".

**ex.:**

```
10 print fn Kubik(22.5, 2*pi, "Kubikzentimeter")
```

Schreibt man `fr` statt `fn` wird die Funktion im [ROM](#) aufgerufen.

## fn Attrib()

```
fn Attrib("Filename")
```

Liefert das Datei-Attribut für die Datei oder das Verzeichnis mit dem Namen `Filename` auf der SD-Card. Das Attribut ist ein Byte-Wert, bei dem jedes Bit für eine Datei-Eigenschaft steht:

Bit-Nr.	Attribut	(wenn gesetzt)	Wert
0	<code>read only</code>	(nur lesen)	1
1	<code>hidden</code>	(versteckt)	2
2	<code>System</code>	(System)	4
3	<code>Volume Label</code>	(Laufwerksbezeichnung)	8
4	<code>Directory</code>	(Verzeichnis)	16
5	<code>Archiv</code>	(neu/geändert)	32

**ex.:**

```
10 if fn Attrib("INC") and 16 then...      prüft auf Verzeichnis
```

## fn Choice()

```
fn Choice(Ausdruck1, Ausdruck2, Ausdruck3)
```

Ist `Ausdruck1` wahr ( $<>0$ ) ist das Ergebnis der Funktion `Ausdruck2`.

Ist `Ausdruck1` unwahr ( $=0$ ) dann ist das Ergebnis `Ausdruck3`:

**ex.:**

```
10 let A=5 : let B=4
20 print fn Choice(A>B, A, B)      druckt den grösseren Wert
```

## fn Dir()

```
fn Dir("Filename", Medium [, Filetyp] )
```

Medium: 0/1=Flash, 2=SD-Card  
filetyp=1 Programm  
filetyp=2 sequentielle Datei

**ex.:**

```
10 print fn Dir("Testfile", 1, 1)      prüft auf Programm
```

Auf dem Flash-Chip liefert `fn Dir()` die Verzeichnis-Position der Datei mit Namen `Filename`. Der `Filetyp` sagt der Funktion, ob sie nach

einer Programm- oder Daten-Datei suchen soll. Wird *kein Filotyp* angegeben, sucht die Funktion nach dem ersten Auftreten des Namens. Ist der Eintrag nicht vorhanden, ist das Ergebnis **0**.

Auf SD-Card meldet die Funktion nur, ob die Datei existiert (**true**) oder nicht (**false**). Der *Filotyp* ist bereits im Namen enthalten:

**ex. :**  
10 if fn Dir("Sound-Test.PBAS", 2) prüft Vorhandensein

## fn Eof()

**fn Eof(Filehandler)**

meldet wenn die 'Datei zu Ende' ist. Die Datei wurde zum Lesen geöffnet. Liest man über das Ende hinaus, gibt es eine Fehlermeldung. **fn Eof()** hilft, dies zu vermeiden. (Eof=End of File):

**ex. :**  
10 let A = fn Open("File", 0, 1) Datei zum Lesen öffnen  
20 while fn Eof(A)=0 Ende noch nicht erreicht?  
30 input #A, A\$ : print A\$  
40 wend  
50 close #A

Das Ergebnis von **fn Eof()** ist True, wenn das Datei-Ende erreicht ist, sonst ist es Null. Siehe [fn Open\(\)](#).

## fn Fileptr()

**fn Fileptr(Filehandler)**

liefert den Filepointer der geöffneten Datei. Also den Zeiger auf die (Byte-)Position, die als nächstes beschrieben oder gelesen wird:

**ex. :**  
10 let A = fn Open("File", 0, 1) Datei zum Lesen öffnen  
20 while !fn Eof(A)  
25 print fn Fileptr(A) Filepointer ausdrucken  
30 input #A, A\$ : print A\$  
40 wend  
50 close #A

## fn Fourier()

**fn Fourier(A0, A1, A2, A3, A4, ....A15)**

füllt den Sound-Samplespeicher mit einer Ton-Schwingung, die einen Grundton und seine ganzzahligen Harmonischen enthält. (ähnlich der Fourier-Synthese einer Hammondorgel). **A0** ist die Amplitude des Grundtons, **A1** die Amplitude des 1. Obertons mit doppelter Frequenz, **A2** die Amplitude des 2. Obertons mit 3-facher Frequenz, **A3** ist die Amplitude des 3. Obertons mit 4-facher Frequenz usw.

```
set sound
sound 0, 220
let $Dummy = fn Fourier(5, 3, 2, 1, 1) Dummy-Ausdruck
```

Prinzipiell müsste die Summe der Amplituden der Grund- und Obertöne kleiner als 1.0 sein. Die Funktion enthält aber eine

"Verstärkungsregelung", sodass beliebige Werte eingesetzt werden können. Das Amplitudenverhältnis bestimmt den Klang. Damit der Grundton vom Gehör als solcher erkannt wird, sollte er die grösste Amplitude haben. **fn Fourier** liefert als Funktionswert den Korrekturwert der "Verstärkungsregelung". (`print $Dummy`)

### fn Fmod()

**fn Fmod(Phasenverschiebung, Faktor)**

füllt den Sound-Samplespeicher mit einer Ton-Schwingung, die einen Grundton und seine ganzzahligen Harmonischen enthält (ähnlich der 1-stufigen FM-Synthese). Phasenverschiebung bestimmt, wie hoch der Anteil der Obertöne ist. Hörbar ab etwa > **20. Faktor** bestimmt die Obertöne Werte: **2...9**:

```
set sound
sound 0, 220
let $Dummy = fn Fmod(90, 5)                                Dummy-Ausdruck
```

### fn Midi()

**fn Midi(MIDI-Noten-Nummer)** oder **fn Midi(Noten-Name)**

liefert **a)** die Frequenz zur **MIDI-Noten-Nummer**. Kann zur Verwendung mit der Anweisung **sound** benutzt werden. Wertebereich: **12...107**

```
set sound
sound 0, fn Midi(60)
```

alternativ kann **b)** auch die **Notenbezeichnung** angegeben werden:

```
set sound
sound 0, fn Midi("F4")
```

auch in einer Stringvariablen:

```
set sound
let A$ = "G#3"
sound 0, fn Midi(A$)
```

Noten: **C, C#, D, D#, E, F, F#, G, G#, A, A#, B**

das **#**-Zeichen erhöht die Note um einen Halbton (**C#** = Cis/Des). Die Ziffer ist die Oktaven-Nummer (**0...7**): der Kammerton A ist in der 4. Oktave (**A4, 440Hz**), das hohe C in der 6. Oktave. Die deutsche Notation wird nur teilweise unterstützt: statt 'B' kann man auch 'H' schreiben.

Tiefste Note: **C0** (16Hz),

höchste Note: **B7** (3951Hz)

## fn Open()

let Var = fn Open("FileName", FileType, Medium) öffnet Dateien

FileType: 0=read, 1=write/create, 2=Append  
Medium: 0/1=Flash-Chip, 2=SD-Card

Es können 11 Dateien auf dem Flash-Chip bzw. 4 Dateien auf der SD-Karte gleichzeitig geöffnet werden. Auf der SD-Karte müssen die Dateien geschlossen sein, bevor man aus ihnen lesen kann. Auf dem Flash-Chip kann man soeben geschriebene Daten sofort lesen.

z.Z. können Dateien zum Lesen (read) Filetyp=0, zum Schreiben (write) Filetyp=1 (erzeugt die Datei auf dem Medium) und zum nachträglichen Anhängen von Daten (append) Filetyp=2 geöffnet werden. Die Variable Var (A) enthält den Filehandler:

```
ex.:
10 let A = fn open("Testfile", 1, 1) erzeugt Testfile
20 print #A, "Dies ist ein erster Test" schreiben
30 close #A schliessen
40
50 let B = fn open("Testfile", 0, 1) öffnet zum Lesen
60 input #B, A$ : print A$ lesen und Drucken
70 close #B schliessen
80
90 let C = fn open("Testfile", 2, 1) öffnet zum Anhängen
100 print #C, "Hier kommen weitere Daten" schreiben
110 close #C schliessen
```

Dies ist ein einfaches Beispiel, das den prinzipiellen Einsatz von **fn Open()**, **close**, **print #** und **input #** zeigt. Eine Datei kann gleichzeitig zum Schreiben und zum Lesen geöffnet sein – mit verschiedenen Filehandlern (Nur Flash-Chip). Eine Datei kann mehrfach zum Lesen geöffnet sein (mit verschiedenen Filehandlern), aber nur einmal zum Schreiben.

**print #** 'drückt' quasi in die Datei (print-Umleitung) genau wie **print** auf den Bildschirm – verkettete Stringelemente werden am Stück geschrieben. **input #** liest solche Stringketten dann auch in einem 'Rutsch' ein.

Zahlen werden mit **print #** ebenfalls als Strings gespeichert. Sie sollten einzeln 'geprintet' werden (will man sie mit **input #** in numerische Variablen einlesen), und nicht direkt hintereinander nur mit Kommata oder Semikolon trennt.

Mit **write #** und **read #** kann man Zahlen sehr kompakt in Dateien speichern und wieder auslesen. (Jede Zahl belegt 8 Bytes, ohne jegliche Separatoren):

```
ex.:
10 let $Zahlen_w = fn open("Zahlen", 1, 1)
20 let $Zahlen_r = fn open("Zahlen", 0, 1)

30 write# $Zahlen_w, 66.5, 227.4, 180.4, 1049.5
40 read# $Zahlen_r, A, B, C, D : print A, B, C, D

50 close# $Zahlen_w, $Zahlen_r
```

**write#** und **read#** gehören zusammen ebenso **print#** und **input#**. Gemischtes Doppel funktioniert nicht: z.B. schreiben mit **write#** und lesen mit **input#**. Kein Problem ist es, nacheinander mit den verschiedenen Anweisungen zu schreiben, wenn beim Lesen mit

dem richtigen Partner gearbeitet wird.

Mit dem Anweisungspaar `put#` und `get#` schreibt / liest man Bytes analog zu den obigen Beispielen. (`get#` liest auf low-level-Niveau jede Art von Daten, die Interpretation obliegt dem User, spannende Beispiele folgen demnächst) siehe Beispielprogramm [Bitmap.PBAS](#)

Siehe auch: [close#](#), [get#](#), [put#](#), [read](#), [write](#), [restore#](#), [dir\\$\(\)](#), [fn Dir\(\)](#), [fn Eof\(\)](#), [fn Fileptr\(\)](#), [dump "DateiName"](#), sowie Beispieldatei: [Open-Test.PBAS](#)

### fn Peek()

holt den Inhalt einer Speicherstelle im RAM oder im ROM. `fn Peek()` kann - anders als [peek\(\)](#) - direkt verschiedene Zahlentypen interpretieren (ohne Konfiguration).

#### fn Peek(32-Bit-Adresse, "Typ")

Typ:	" <b>s</b> "=Short,	" <b>b</b> "=Byte,	8Bit
	" <b>i</b> "=Integer,	" <b>w</b> "=Word,	16Bit
	" <b>n</b> "=Longint,	" <b>l</b> "=Longword,	32Bit
	" <b>f</b> "=Float32,	" <b>x</b> "=Float64	Fliesskomma

`fn Peek()` ist ca 30% langsamer als `peek()`.

### fn Pointer()

#### fn Pointer(32-Bit Adresse)

liefert einen Zahlenwert, der angibt, welchen Inhalt die Adresse hat.

1=numerische Variable (A...Z), Float64  
2=Namensvariable, Float64  
3=Array A()...J() Element, Float64  
4=Array L Element, Long32  
5=Array P Element, Long32  
6=Array S Element, 16Bit Integer  
7=Array Z Element, Float64  
8=Stringvariable A\$...H\$  
9=Textarray-String  
11=Programmzeile  
12=function Label  
13=sub Label  
14=proc Label  
15=Default Byte8  
16=Default Word16  
17=Default Long32  
18=Default Float64

### fn Rgb()

`fn Rgb(Rot, Grün, Blau [, 16 | 24 | 1])`  
`fn Rgb("Farbname" [, 1])`

Die erste Syntax liefert einen 24-Bit oder 16-Bit Farbwert aus den 3 Farb-Anteilen.

`Rot`, `Grün` und `Blau` können Werte von 0...255 annehmen (RGB888) bzw. 0...31 oder 0...63 (RGB565). Werden nicht geprüft aber auf 5-/6- oder 8-Bit-Wert zurechtgestutzt. Der 4. Parameter ist die Bitzahl (`16` oder `24` Bit) bzw. `1`, dann orientiert sich die Funktion am TFT-LCD. Keine Angabe bedeutet 24 Bit.

Die 2. Syntax liefert die *Farbnummer* (Index) der Farbpalette für die 23 vordefinierten Farbnamen. Wird der 2. Parameter mit **1** zusätzlich angegeben, ist das Ergebnis der 24Bit bzw. 16Bit *Farbwert* (wie er im Paletten-Array **P()** steht).

fn Rgb2()

fn **Rgb2(Rot, Grün, Blau)**

liefert einen 24-Bit Farbwert aus den 3 Farb-Anteilen.

**Rot, Grün** und **Blau** können Werte von 0...255 annehmen (RGB888). Werden nicht geprüft aber auf Byte-Wert zurechtgestutzt. Der Unterschied zu **fn Rgb()** liegt im 24-Bit-Ergebnis, bei dem Rot und Grün vertauscht sind. (einige Neopixel LEDs benötigen diese vertauschte Anordnung der Farbwerte)

fn Red()

fn **Red(24-Bit- | 16-Bit-Farbwert [, 24 | 16 | 1])**

liefert den Rot-Anteil eines **24**-Bit oder **16**-Bit Farbwerts. Der 2. Parameter ist die Bit-Angabe. Ist der Farbwert **24**Bit, dann hat der Rot-Anteil 8Bit. Ist der Farbwert **16**Bit, dann hat der Rot-Anteil 5Bit. Wird eine **1** als 2. Parameter angeben, orientiert sich die Funktion am TFT-LCD. Keine Bit-Angabe bedeutet **24**Bit. Siehe auch Farbpalette.

print fn Red(4197124)	Ergebnis: 64
-----------------------	--------------

fn Green()

fn **Green(24-Bit- | 16-Bit-Farbwert [, 24 | 16 | 1])**

liefert den Grün-Anteil eines **24**-Bit oder **16**-Bit Farbwerts. Der 2. Parameter ist die Bit-Angabe. Ist der Farbwert **24**Bit, dann hat der Grün-Anteil 8Bit. Ist der Farbwert **16**Bit, dann hat der Grün-Anteil 6Bit. Wird eine **1** als 2. Parameter angeben, orientiert sich die Funktion am TFT-LCD. Keine Bit-Angabe bedeutet **24**Bit. Siehe auch Farbpalette.

print fn Green(33024,16)	Ergebnis: 8
--------------------------	-------------

fn Bue()

fn **Blue(24-Bit- | 16-Bit-Farbwert [, 24 | 16 | 1])**

liefert den Blau-Anteil eines **24**-Bit oder **16**-Bit Farbwerts. Der 2. Parameter ist die Bit-Angabe. Ist der Farbwert **24**Bit, dann hat der Blau-Anteil 8Bit. Ist der Farbwert **16**Bit, dann hat der Blau-Anteil 5Bit. Wird eine **1** als 2. Parameter angeben, orientiert sich die Funktion am TFT-LCD. Keine Bit-Angabe bedeutet **24**Bit. Siehe auch Farbpalette.

print fn Blue(33024,16)	Ergebnis: 0
-------------------------	-------------

fn Size()

fn Size("Filename", Medium [, Filetyp] ) Medium: 0/1=Flash, 2=SD-Card  
filetyp=1 Programm  
filetyp=2 sequentielle Datei

ex. :

```
10 print fn Size("Testfile", 1, 1)      Programmgrösse
```

liefert die Dateigrösse der Datei mit Namen **Filename**. **Filetyp** kann nur auf dem Flash-Chip angegeben werden. Ohne die Angabe **Filetyp** wird die erste Datei mit Namen **Filename** gewählt.

fn Testbit()

überprüft, ob ein Bit gesetzt ist oder nicht.

fn Testbit(Rechen-Ausdruck, Bitnummer)

anders als der Dezimalpunkt kann diese Funktion jeglichen numerischen Ausdruck in ein 32Bit Longword verwandeln und auf ein gesetztes Bit prüfen. Ergebnis ist 0 oder 1:

```
10 if fn Testbit(sys(11), 7) then...    prüft, ob automatische  
                                         Sommerzeitumschaltung inaktiv ist
```

high()

liefert das höhere **16-Bit-Word** einer **32-Bit-Long-Word-Zahl**.

```
print high(99999)                      Ergebnis: 1
```

int()

Liefert den Ganzahlwert eines Real-Wertes zurück. Nachkommastellen werden abgeschnitten.

```
print int(-9.5)                        Ergebnis: -9
```

low()

liefert das niedere **16-Bit-Word** einer **32-Bit-Long-Word-Zahl**.

```
print low(99999)                      Ergebnis: 34463
```

log()

berechnet den natürlichen Logarithmus zur Basis e (auf einigen Taschenrechnern als **ln()** bezeichnet):

```
10 print log(2.72)                    Ergebnis: 1.00063  
20 let B = log(e+1.1)
```

den 10-er Logarithmus erhält man, indem man durch **log(10)** teilt:

```
print log(5)/log(10)                ergibt 10-er Logarithmus von 5
```

## norm()

berechnet näherungsweise den Korrektur-Faktor für den normierten Luftdruck für die angegebene Höhe über Normal-Null.

```
print 960/norm(100)          Ergebnis: 971.46276
```

Der Korrekturfaktor für 100m Höhe über dem Meeresspiegel, hier für einen gemessenen Luftdruck von 960 mbar. Durch diesen 'Faktor' muss der Messwert geteilt werden. Siehe auch [sensor\(\)](#).

## not()

bitweises not, das Ergebnis ist ein **32 Bit-Long-Word-Wert**.

**Not(0) = 4294967295** (-1 als Integer in anderen Basic-Dialekten)

```
ex.:
10 let A = not(B)
20 print not(1+C)
30 let A.2 = not(A.2)           Bit Invertierung
```

Siehe auch [Operator '!'.](#)

## num()

überprüft einen **String**, ob [calc\(String\)](#) keine Fehlermeldung erzeugen würde - dann ist das Ergebnis **true** - Fehlermeldung ergibt **false**:

```
ex.:
10 let A$ = #3+(9*2)#
20 print num(A$)              Ergebnis: 4294967295
```

Leere Strings haben **0 (false)** als Ergebnis, [calc\(\)](#) und [val\(\)](#) melden aber z.Z. (V1.05) keine Fehler.

## peek()

Liest den Inhalt einer Speicherzelle (**Byte**, **8-Bit**) aus. Es ist der gesamte 32Bit-Adressraum - RAM, ROM, Register - erreichbar. Siehe auch [poke\(\)](#).

```
print peek(@A+1)
```

Es können auch Longwords (32Bit ohne Vorzeichen), Word (16Bit ohne Vorzeichen) und Float (64Bit)-Werte gelesen werden. Siehe [config](#). Siehe auch [fn Peek\(\)](#).

## sensor()

[sensor\(\)](#) liefert Messwerte verschiedener Sensoren am Sensor-Bus oder am One-Wire-Bus.

\*

### 1.) HTU31-Sensor:

[sensor\(0\)](#) liefert die Temperatur und [sensor\(1\)](#) die Luftfeuchte des HTU31-Sensors. Die Abfrage dauert 100ms. Siehe auch [i2c](#).

```
ex.:
10 print "Temperatur: "; sensor(0); "°C"
```

- \* 2.) BME280 / BMP280-Sensor:  
`sensor(2)` liefert den Luftdruck, `sensor(3)` die Temperatur und `sensor(4)` die Luftfeuchte. `sensor(4)` funktioniert nur beim BME280.

**ex.:**  
`10 print "Luftdruck: "; sensor(2)`

oder wenn der normierte Luftdruck (auf Meereshöhe) angezeigt werden soll, wie vom Wetterdienst:

**ex.:**  
`10 print "Luftdruck: "; sensor(2) / norm(38)`

`38` ist die Höhe (in Metern) über NN von Düsseldorf, meinem Heimatplaneten. Hier bitte die Höhe Deiner Stadt eingeben. Siehe auch [norm\(\)](#).

- \* 3.) DS18B20 (DS18S20) One-Wire-Temperatur-Sensor:

Wenn nur 1 - 2 Sensoren benötigt werden:

`convert 6` startet die Temperatur-Messung an Port PA6 (Pin28/J3)  
`sensor(6)` liefert die Temperatur des Sensors an Port PA6 (Pin28/J3)

`convert 7` startet die Temperatur-Messung an Port PA7 (Pin27/J3)  
`sensor(7)` liefert die Temperatur des Sensors an Port PA7 (Pin27/J3)

**ex.:**  
`10 convert 6 : convert 7` startet Konversion  
`20 wait 800` zwischen Start und Abruf müssen mindestens 750ms liegen, der Computer kann in dieser Zeit auch etwas anderes machen als nur zu warten.  
`30 print sensor(6)` 1.Sensor auslesen  
`40 print sensor(7)` 2.Sensor auslesen

Es werden mehr als 2 Sensoren benötigt?

In diesem Fall werden alle Sensoren an Port PA6 (Pin28/J3) angeschlossen. (Mit nicht-parasitärer Beschaltung! \*)

Vorher muss aber die Serien-Nummer jedes Sensors **einzelne** ausgelesen werden. Dazu schliesst man den Probanden an Port PA7 (Pin27/J3) an:

`sensor$` liefert die 64Bit-Serien-Nr. des Temperatur-Sensors als String:

<pre>&gt; print sensor\$</pre> <pre>Temperature Sensor is DS18B20 6630167858422087464 ready</pre>	<b>Eingabe</b>  <b>Systemantwort</b> <b>Seriennummer</b>
---	---

Die Serien-Nummer bitte notieren (evtl. auch Sensor beschriften).

Sind die Serien-Nummern alle bekannt, können die Sensoren an PA6 (Pin28/J3) angeschlossen werden. Nun kann man die Temperatur jedes Sensors mit seiner Serien-Nummer auslesen:

5 let A\$ = "6630167858422087464" 10 convert A\$ 20 wait 2000 30 print sensor(A\$)	Serien-Nummer startet Messung hat sich bewährt liest 1 Sensor aus
---	--

oder so:

10 do 20    restore 20    for N=1 to 3 30    read A\$ 40    convert A\$ 70    print fix\$(sensor(A\$), 12); 80    next 90    print chr\$(13); 95 until 100 data "6630167858422087464", "10016014381027080464" 120 data "12682145352766479376"	Zeiger auf 1. Serien-Nummer 3 Sensoren liest Serien-Nummer ein startet Messung Messwertausgabe Überschreiben Serien-Nummern
---	---

Rein theoretisch sind ca. 100 One-Wire-Sensoren und ca. 100m Leitungslänge möglich... (getestet und garantiert mit 6 Sensoren).

\*) 3 Draht-Leitung: Plus 3,3V, Daten, Ground (Masse, Minus), während der Messung werden 1,5mA je Sensor benötigt. Siehe [Anhang](#).

Während der Kommunikation mit den Sensoren werden on-Interrupts abgeschaltet.

Wurden die entsprechenden I2C-Sensoren beim Start-Scan des Basic-Computers nicht entdeckt, wird mit Fehlermeldung abgebrochen.

### sign()

Vorzeichenfunktion. Die Funktion liefert 1 bei allen Real-Werten > 0, -1 bei Werten < 0 und 0, wenn der Wert 0 beträgt.

let A = sign(3.5)	Ergebnis: A = 1
-------------------	-----------------

### sin()

berechnet den Sinus eines Radiant-Wertes

? sin(.234) print sin(pi/8)+B)	irgend eine Formel
-----------------------------------	--------------------

### sqr()

berechnet die Quadratwurzel eines (positiven) Real-Wertes

? sqr(13.56) let A = sqr(P/2)
----------------------------------

## sys()

meldet verschiedene System-Werte, die während des Betriebs auftreten. Siehe [Anhang](#) oder mit `dump 9` auflisten.

```
ex.:
10 let A = sys(6) : print "Terminal-Cursor-Position:"
20 print "Zeile "; low(A), print "Spalte "; high(A)
30 if sys(8)
40   print "Sommerzeit"
50 else
60   print "Winterzeit"
70 endif

ex.:
10 do : print sys(24); chr$(13); : until
20 rem zeigt Anzahl der Zeichen im USB-Empfangspuffer an
```

## tan()

berechnet den Tangens eines Radian-Wertes

```
? tan(.234)
print tan(pi/4)
```

## val()

`val(Stringvariable)`

`val()` liefert den **numerischen** Wert, den der *Inhalt* von **Stringvariable** repräsentiert. Der *Inhalt* von **Stringvariable** muss eine Zahl sein. Ist der Inhalt von **Stringvariable** nicht **numerisch**, ist das Ergebnis Null. Siehe auch [calc\(\)](#).

```
ex.:
10 let A$ = "125.45"                                Dezimalzahl (keine Spaces)
20 let A = val(A$)
30 let B$ = "0x3FFA"                                 Hexadezimal- oder Binärzahl
40 print val(B$)
```

`val()` kann auch mit den Elementen des **Text-Arrays** umgehen:

```
10 let T(0) = "3.14159"                            T(0) ist Text 0 im Textarray
20 print val(T(0))
```

`val()` kann selbst auch wieder in mathematischen Ausdrücken benutzt werden:

```
let A = 3*val(A$)+5
```

`val()` funktioniert auch mit Teilstrings:

```
10 let A$ = "Mo, 20.06.2022"
20 let B = val(A$(8, 2))                           Teilstring: "06"
30 print B                                         Ergebnis: 6
```

## User definierte Funktionen1

Es gibt **5** User-definierte-Funktionen `fn1()` ...`fn5()` vom **Typ1**. Mit der Anweisung `deffn` wird ein zusammenhängender Programm-Zeilen-Block als Funktionskörper erklärt. Im Programmblöck werden die Anweisungen ausgeführt, die zum Funktionsergebnis führen. Die Funktion selbst kann wie jede numerische Funktion in Rechenausdrücken verwendet werden:

```
10 rem Hauptprogramm
20 deffn Quadrat() as 1
...
50 print 3+fn1(2.75)
...
120 end.

1000 function Quadrat
1010 let result = fnvar*fnvar
1020 end
```

definiert die Funktion Quadrat als Funktion Nr.1  
aufgerufen wird die Funktion als `fn1()` in einem Rechenausdruck  
**Ergebnis: 10.5625**

Programm-Ende

ab hier wird die Funktion 1 abgearbeitet. Es können alle Anweisungen verwendet werden.

fnvar und result sind Variablen für den Austausch mit der Funktion. `fnvar` ist das Argument und `result` das Ergebnis (Resultat). Hier berechnet sie das Quadrat des Arguments.

die Funktion ist fertig

Bei Aufruf der Funktion `fn1()` (im Rechenausdruck Zeile 50) wird das *Funktionsargument* (hier: `2.75`) an die Variable `fnvar` übergeben. Mit ihrem Wert kann man nun rechnen, steuern, messen etc. Der Rechenausdruck wartet jetzt auf das *Ergebnis* der Funktion `fn1()`, deren Berechnung im Funktionsblock ab *Label function Quadrat* geschieht.

Vor Verlassen des Funktionsblocks in Zeile 1020 mit `end` übergibt man das *Resultat* an die Variable `result`, das nun seinerseits im Rechenausdruck weiterverwendet wird. Ohne eine Zuweisung `let result = Ausdruck` ist das *Resultat* Null.

Die Funktionen können aber nicht nur rechnen, sondern sämtliche Anweisungen ausführen. Z.B. Messwerte, die durch eine Anweisung erzeugt wurden, als Funktionswert zurückgeben oder Eingänge abfragen, Relais einschalten und Rückmeldung geben usw. Der Funktionsblock muss mit `end` formal beendet werden. Zusätzlich darf `end` oder noch besser: `exit` auch wie ein Ausgang aus dem Block wo nötig eingesetzt werden.

### Lokale Variablen

Damit es keinen Namenskonflikt mit den Variablen des Hauptprogramms gibt, hat jede User-Funktion einen eigenen kleinen Satz einfacher numerischer Variablen sowie zwei Stringvariablen. Diese Variablen (`A-J` und *neu*: die Stringvariablen `A$` und `B$`) sind nur **lokal** sichtbar in der Funktion und **unabhängig** von den **globalen** Variablen `A-J` sowie `A$` und `B$` im Hauptprogramm. Die Variablen `K-Z` und Namen-Variablen können natürlich auch verwendet werden z.B. zum Austausch mit dem Hauptprogramm.

Die Funktionen haben auch einen eigenen Lesezeiger für `read`.

### Rekursion

Eine User-Funktion sollte sich nicht selbst aufrufen. Man müsste sonst bei jedem Selbstaufzug den Taskzähler überprüfen, ob er noch nicht `10` erreicht hat. Abfragen mit `sys(0)`. Mehr als `9` User-Prozeduren/-Funktionen kann man z.Z. nicht verschachteln.

## User definierte Funktionen2

```
ex.:
10 rem Hauptprogramm

50 print fn Kubik(7.2, 8.3, 6.4)
...
120 end.
...
...
1000 function Kubik(A, B, C)
1010 let result = A*B*C
1020 end
```

aufgerufen wird die Funktion als **fn Kubik()** in einem Rechenausdruck **Ergebnis: 382.46398**

Programm-Ende

Label **Kubik**, hier startet die Funktion

**result** ist das Ergebnis der Funktion.

die Funktion ist fertig

Bis zu 16 Argumente können diese Funktionen aufnehmen und sie benötigen keine Definition mit **deffn**, der Funktionsname **fn Name()** muss mit dem Labelnamen **function Name** übereinstimmen. Siehe auch [function](#).

Diese 2. Art Funktionen starten etwas langsamer als die Funktionen vom Typ 1, da sie die Adresse des Labels bei jedem Aufruf neu ermitteln. In Interruptroutinen können Timing-Probleme auftreten.

Die Argumente des **Funktionsaufrufs** im Rechenausdruck (numerisch oder auch Strings) werden eins zu eins in die Variablenliste des **function**-Labels übertragen. Es gibt keine Argumentvariable **fnvar** aber eine Resultatvariable **result**. Siehe auch [User-Stringfunktionen](#).

Schreibt man **fr** statt **fn** für den Funktionsaufruf wird die Funktion im [ROM](#) aufgerufen.

Ein praktisches Beispiel gefällig? Wenn man noch eine freie serielle Schnittstelle hat, kann man einen Sensor ja vielleicht auch dort anschliessen:

```
10 proc MHZ19B-Test
20   input "Schnittstelle?: ", $Interface
30   select $Interface
40     case 1 : config com 1, 9600
50     case 2 : config com 2, 9600
60     case 3 : config com 3, 9600
70     case 4 : config com 4, 9600
80     celse : print "falsches Interface!" : exit.
90   selend
100  do
110    print "CO2: "; str$(fn MHZ19B($Interface),5); " ppm"; chr$(13);
120    wait 1000
130  until
140 end.
150

1000 function MHZ19B(G)
1005   clear buf G
1010   put #G, 255, 1, 134, 0, 0, 0, 0, 0, 121
1020   get #G, A, B, C, D, E, E, E, E, F
1030   if (A=255) and (B=134)
1040     let result = 256*C+D
1050   else
1060     let result = 0
1070   endif
1090 end
```

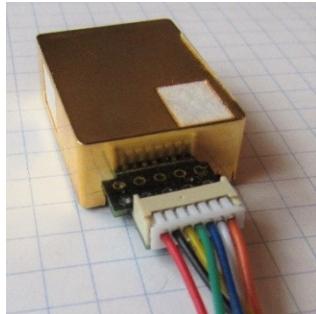
**Hauptprogramm**  
Eingabe der ser. Schnittstelle  
für Sensor konfigurieren

**Hauptschleife**  
1 Sekunde warten

**Funktion MHZ19B**  
Empfangspuffer leeren  
Startkommando senden  
9 Bytes auslesen  
Rahmen korrekt?  
High- & Low-Daten-Byte

Rahmenfehler

Das Beispiel liest einen CO<sub>2</sub>-Sensor MH-Z19B aus, der an einer der seriellen Schnittstellen hängt. Für eine Funktions-Bibliothek eignen sich nur die Funktionen vom Typ2. Die Funktion **MHZ19B()** liefert den CO<sub>2</sub>-Messwert oder 0, wenn etwas nicht in Ordnung ist (der Messwert ist niemals 0), übergeben wird nur die Nummer der Schnittstelle in **G**. Serielle Schnittstelle 1 nur verwenden, wenn die Konsole am USB betrieben wird.



Verbunden wird der Sensor wie folgt:  
(serielle Schnittstelle 2)

Draht	Signal	Steckerleiste
blau:	mit Gnd	J2 / 6
grün:	mit +5V	J2 / 2
gelb:	mit PC10	J2 / 25 (TX MIDI)
schwarz:	mit PC11	J2 / 28 (RX MIDI)
orange, weiss, rot nicht verbinden. Sollten die Farben abweichen, Drahtposition stattdessen heranziehen.		

[siehe Anschlußplan](#)

Abb. MH-Z19B CO<sub>2</sub>-Sensor

# Stringfunktionen

( )

("A") liefert den ASCII-Wert vom Buchstaben A. Wie die Funktion `asc()` in Vintage-Basic-Dialekten.

```
let A = ("A")          weist der Variablen A den ASCII-Wert des Buchstabens A zu (65)
```

bei Zeichenketten liefert die Klammer () den ASCII-Wert des 1. Zeichens des Strings.

```
print ("Hallo Welt")      liefert als Ergebnis: 72
```

Funktionsklammern liefern für stringbezogene Funktionen wie: `asc()`, `epoch()`, `f$()`, `len()` usw. sowie für die Userfunktionen vom TypII, die Adresse des Strings.

`asc()`

`asc(Zeichen)`

aus Kompatibilitätsgründen gibt es zusätzlich die Funktion `asc()`. `asc()` liefert die ASCII-Ordnungs-Nummer eines Zeichens.

```
let A = asc("A")
```

`bin$()`

`bin$(Zahl [, Breite] )`

liefert eine Zeichenkette, die den Wert `Zahl` binär darstellt. `Zahl` ist eine 32-Bit-Wort-Zahl. Das Zeichen «%» kennzeichnet eine Binärzahl. Wird die String-`Breite` (Stellenzahl) angegeben, fällt das Literal-Kennzeichen «%» weg. `Breite` sollte der zu erwartenden Stringbreite mindestens entsprechen (max.32).

```
print bin$(4+333)      Ergebnis: %101010001 ohne Angabe der Breite
print bin$(4+333, 12)   Ergebnis: 000101010001 Breite 12
```

`chr$()`

`chr$(Zahl)`

liefert ein Zeichen mit der ASCII-Nummer `Zahl`. `Zahl` ist ein Bytewert. Nicht alle Werte (0 ... 255) ergeben sichtbare Zeichen (`Zahl > 31`).

```
print chr$(13)          sendet Wagenrücklaufzeichen (Return)
```

```
ex.:
10 for N=1 to 7
20   for M=0 to 31
30     print chr$(N*32+M);
40   next M
50   print
60 next N
```

Schreibt den Zeichensatz auf das Terminal:

cls\$

erzeugt einen String, der den VT100-Steuercode zum Löschen des Terminal/Konsolen-Fensters enthält. Für Meldungen aus Interrupt-Routinen, in denen direkt nicht gedruckt werden kann oder für Ausgaben mit `print#`.

```
ex.:
10 let A$ = cls$ & "pahlbasic"
20 print A$
```

löscht das Terminalfenster und schreibt **pahlbasic** in die linke obere Ecke, exakt wie:

```
ex.:
10 let A$ = "pahlbasic"
20 cls : print A$
```

color\$()

color\$(Zeichenfarbe [, Hintergrundfarbe])

**color\$()**bettet die VT100-Steuercodes für die Schriftfarbe auf dem Konsolenfenster in einen String ein. **Zeichenfarbe** 0...15, **Hintergrundfarbe** 0...7.

Die VT100 Farben siehe [Anhang](#). Die Farbeinstellung gilt nur für den jeweiligen String – auch wenn das Senden der print-Farben mit `set print color=16` abgeschaltet wurde.

Die Systemvariable gleichen Namens (`color`) bezieht sich *nur* auf das LCD. Default Zeichenfarbe=7, Hintergrundfarbe=0 (schwarz):

```
ex.:
10 let A$ = cls$ & color$(13,0) & "pahlbasic"
20 print A$
```

```
> run  
pahlbasic  
ready  
> _
```

count\$( )

count\$(Zahl [ , Breite ] )

`count$` erzeugt einen String aus `Zahl` im Stil einer mechanischen Zähleranzeige. `Zahl` ist eine positive 48-Bit-Ganz-Zahl. `Breite` gibt die Zähler-Stellenzahl an. Wird `Breite` nicht angegeben ist die Stellenzahl 10.

```

print count$(55)          Ergebnis: 0000000055
print count$(55, 8)        Ergebnis:    00000055

```

dir\$

dir\$  
dir\$(Datei-Nummer) Datei-Nummer=Nummer des Directory-Eintrags

1.) aktuelles Verzeichnis auf der SD-Card

```

print dir$                  Volume:\ 

```

2.) Dateiname, wie er im Directory des Flash-Chips eingetragen ist:

```

print dir$(8)                Zeiten-Tabelle

```

dow\$

Wochentag als String:

```

print dow$                  heutiger Wochentag
print dow$(3)                Mittwoch
                                            auch: print dow$(dow)

```

seit V1.06B2 auch englische Wochentage, wenn config start !=1 konfiguriert wurde.

epoch\$()

epoch\$(UNIX-Time)

```

ex.:                               Ergebnis:
10 print epoch$(1660478110)      14.08.2022,11:55:10

```

Unix-Zeit als Datum/Uhrzeit-String. Gegenstück zur epoch() Funktion.

f\$, fs

let Stringvar = f\$ Funktionsname(Argument1, Argument2, Argument3 ,...)

User-Stringfunktion, funktioniert exakt wie die numerischen User-Funktionen vom TypII. Das Resultat ist aber ein String: result\$. Es gibt nur zwei *lokale* Stringvariablen: A\$ und B\$.

```

ex.:
10 proc Demo-Stringfunktion
20 let H$ = "hausmeister"
30 print f$ Kapital(H$, 1)                                     globale Variable H$
90 end.
95
1000 function Kapital(A$, A)                                     schreibt Wort gross
1010   if (A<1) or (A>len(A$)) then exit
1020   if (asc(A$(A))>96) and (asc(A$(A))<123)                 lokale Variable A$
1030     let A$(A) = asc(A$(A))-32
1040     let result$ = A$
1050   endif
1900 end

```

```

run
Hausmeister
ready

```

**fs** funktioniert wie **f\$**, der Funktionsblock selbst befindet sich aber im [ROM](#).

```
fix$()  
fix$(Zahl [, Breite ])
```

**fix\$()** verwandelt das Ergebnis eines [Rechenausdrucks](#) in einen [String](#). Das [Ergebnis](#) wird vorher auf [2](#) ([3](#)) Stellen hinter dem Komma gerundet. Siehe auch [str\\$](#). Siehe auch [config](#).

```
print fix$(pi)          Ergebnis: 3.14  
print fix$(pi, 8)       Ergebnis:    3.14
```

Soll der erzeugte String eine bestimmte (Mindest-) [Breite](#) haben, kann man diese zusätzlich angeben. [Breite](#) muss mindestens der zu erwartenden Stellenzahl +1 entsprechen. Es werden zusätzliche Spaces eingefügt, bis [Breite](#) erreicht ist. Dies ist für den Ausdruck von unterschiedlich großen Zahlen an derselben Position (Terminal oder LCD) interessant, damit Ziffern von vorherigen Ausdrucken nicht 'stehen bleiben', sondern überschrieben werden.

```
fkey$()  
fkey$(Funktionstasten-Nummer)
```

liefert den Text, mit dem die Funktionstaste (1...12) belegt wurde. Siehe auch [fkey](#).

```
fkey 9, "get A,B,C,D,E" + chr$(13)      wird bei Betätigung sofort ausgeführt  
print fkey$(9)      Ergebnis:      get A,B,C,D,E
```

```
gps$  
gps$  
gps$("Sentence")
```

pahlbasic bietet z.Z. nur rudimentäre GPS-Funktionen an. [gps\\$](#) bietet die Möglichkeit, die NMEA-Strings selbst zu parsen.

1.) [gps\\$](#)

liest den NMEA-Sentence [RMC](#) ein:

```
> print gps$  
$GP$RMC,134659.00,A,5112.23947,N,00646.33937,E,0.533,,280223,,,A*75  
ready
```

2.) [gps\\$\("Sentence"\)](#)

liest den [gewünschten](#) NMEA-Sentence ein. Bitte nur NMEA-Strings wählen, die Ihr GPS-Empfänger auch liefert. Die Bezeichnungen der Sentences sind nur 3 Zeichen lang. Die Sentences beginnen alle mit \$GP gefolgt von der Sentence-Bezeichnung. Beim UBLOX7-Modul sind das: RMC, VTG, GGA, GSA, GSV und GLL:

```
> print gps$("GGA")  
$GP$GGA,135236.00,5112.23481,N,00646.34174,E,1,04,5.19,40.6,M,46.5,M,,*  
ready
```

Im Netz sind jede Menge Informationen zum Aufbau von [NMEA Sequenzen](#). Das nachfolgende Beispiel liest die Anzahl der sichtbaren Satelliten aus:

```

ex.:
10 proc Anzahl_Satelliten
100 print "Anzahl Satelliten: "; fn Satellites()
900 end.
980
990
1000 function Komma(A$, A, D) : ' A=Anzahl Kommas, D=ZeichenOffset
1020 let B = len(A$) : let E = 5 : ' Startposition 5
1030 while E<B
1035 if (A$(E)=",") then inc C : ' C zählt die Kommas
1040 if C=A then break : ' vergleichen mit A
1050 inc E
1060 wend
1070 let result = E+D
1090 end
1100
1110
2000 function Satellites()
2010 let A$ = gps$("GGA") : ' hinter dem 7.Komma 2-Stellig:
2020 let result = ((A$(fn Komma(A$,7,1))-48)*10+(A$(fn Komma(A$,7,2))-48)
2090 end

```

**hex\$()**  
**hex\$(Zahl [, Breite ] )**

liefert eine **Zeichenkette**, die den Wert **Zahl** hexadezimal darstellt. **Zahl** ist eine 32-Bit-Wort-Zahl. Das Zeichen «**0x**» kennzeichnet eine Hexadezimalzahl.  
 Wird die **String-breite** angegeben, fällt das Literal-Kennzeichen «**0x**» weg. **Breite** sollte der zu erwartenden Stringbreite mindestens entsprechen:

print hex\$(3*9)	Ergebnis: 0x1B	ohne Angabe der Ausgabe-Breite
print hex\$(3*9, 6)	Ergebnis: 00001B	mit Breite 6

in

**(String1 in String2)**

liefert als Ergebnis die Position in **String2**, ab welcher **String1** dort vorkommt. Kommt **String1** nicht in **String2** vor, liefert **in** den Wert **0**. **String1** und **String2** dürfen auch Teilstrings sein.

```

ex.:
10 let A$ = "1234567890"
20 let A = ("678" in A$)
30 if ("678" in A$) then print "gefunden an Position: "; A
40 if ("678" in A$)=6 then print "richtige Position"
50 print ("678" in A$)

```

**in** funktioniert auch mit den Elementen des Textarrays. **in** ist eigentlich eine numerische Funktion...

**inkey**

**inkey** holt ein Zeichen von der Konsole (USB oder serielle Schnittstelle Nr.1). Ist kein Zeichen im Empfangspuffer, ist das Ergebnis das Null-Zeichen. **inkey** wartet nicht auf eine Eingabe. **inkey** verhält sich **numerisch** z.B. bei **let A=inkey** oder **print (inkey)** aber wie ein **String** bei **print inkey** und **let A\$=inkey**. Ebenso verhält es sich bei Vergleichen:

```
ex.:
10 if inkey=13 goto 200                                numerisch
10 if inkey="r" goto 200                                String
10 while (A$=="") : let A$ = inkey : wend
20 print A$                                              auch so kann man auf eine
                                                          Eingabe warten (String)
```

```
ex.:
10 do                                                 Hauptschleife
20   do : until buf
30   let A$ = inkey : print A$;
40   let E$(0) = A$
50 until inkey=13                                     wartet auf Eingabe
                                                       Zeichen holen, und ausgeben
                                                       Ergebnis-String bilden, siehe Pos.0
                                                       bis Return gedrückt
```

Das Beispiel ahmt die Basic-Anweisung [input](#) nach. (aber ohne die Editierfunktionen)

## inkey\$

inkey\$ wartet auf ein Zeichen von der Konsole und übergibt es an den ([Teil-](#)) String:

```
ex.:
10 let A$ = inkey$                                     A$ enthält nur das Eingabezeichen
20 let A$(0) = inkey$                                 Eingabezeichen an A$ anhängen
30 let T(9) = string(10, 32)                         String aus 10 Spaces
40 let T(9, 5) = inkey$                             5. Zeichen mit Eingabe überschreiben
```

oder an eine numerische Variable:

```
ex.:
10 let A = (inkey$)                                    A enthält den ASCII-Wert des
                                                       Eingabezeichens
```

In einigen Vintage-Basic-Dialekten verhalten sich inkey und inkey\$ genau anders herum: inkey wartet auf Eingabe und inkey\$ wartet nicht. Bei der Übertragung von älteren Basic-Texten dies bitte beachten. In Interruptroutinen wartet inkey\$ nicht.

## & , + , \_ , \

String Verkettung (hängt mehrere Zeichenketten aneinander)

```
ex.:
10 let B$ = "Hallo" & " Welt"    oder: 10 let B$ = "Hallo" + " Welt"
20 let A$ = date _ clock _ "Uhr"  statt: 30 print A : print B : print C
```

Funktioniert für Zeichenketten, Strings ähnlich wie bei [print](#). Numerische Komponenten müssen aber mit str\$, fix\$ oder hex\$ bzw. bin\$ in Strings konvertiert werden.

Der [Unterstrich](#) \_ fügt einen Tabulator zwischen die Elemente – wie das Komma bei [print](#). Das & verhält sich wie das Semikolon bei [print](#) (hängt die Strings ohne Zwischenraum aneinander). Besteht keine Verwechselungsgefahr mit dem Additionszeichen, kann auch „+“ statt „&“ benutzt werden (z.B. bei String-Literalen in Anführungszeichen, jedoch *nicht* bei [print](#)). Das „\“ fügt ein Newline (Zeilenschaltung) ein – sowohl in Strings als auch bei [print](#).

```
40 print "Ver" & "kettung"                                bildet den Summenstring und druckt ihn
```

hat das gleiche Ergebnis wie:

```
50 print "Ver"; "kettung"
```

druckt die Einzelstrings hintereinander

## lcase

**lcase String**

der **(Teil-)Ergebnis-String** wird in Kleinbuchstaben verwandelt. **lcase** ist ein String-operator.

**ex.:**

```
10 let A$ = "HALLO WELT"  
20 print lcase A$(7, 4)
```

Ergebnis: welt

## len()

**len(String)**

**len()** liefert die Länge eines Strings:

```
print len(A$)  
print len(T (Nummer))  
print len(clock)
```

für die **Stringvariablen** A\$...H\$

für die **Textarray-Strings**

für **Strings ohne Adresse** wie: **date, gps\$ etc.**

## loc\$()

**loc\$(Zeile, Spalte)**

bietet die VT100-Steuercodes für die Cursor-Positionierung in einen String ein, exakt die gleichen Codes, die auch die Anweisung **locate** ausdrückt.

Die Anweisung **locate** funktioniert nicht auf allen Schnittstellen, **loc\$()** schon. Die max. Werte für Zeile und Spalte siehe Anweisung [locate](#).

**ex.:**

```
10 let A$ = loc$(15, 20) & "Temperatur-Alarm"  
20 print A$
```

macht exakt das Gleiche wie:

**ex.:**

```
10 let A$ = "Temperatur-Alarm"  
20 locate 15, 20 : print A$
```

## mid

**mid Zeichenposition, Anzahl Zeichen, String**

beschneidet den **String** auf **Anzahl Zeichen** ab **Zeichenposition**:

```
10 print mid 1, 5, clock
```

druckt die Uhrzeit ohne die Sekunden. **mid** ist eher Operator als Funktion. Negative Werte für Zeichenposition zählen vom rechten String-Ende.

## mpeek()

`mpeek(32-Bit-Adresse [, Anzahl] [, Delim] )`

liest einen Speicherbereich aus und übergibt ihn als String. **Anzahl** = Anzahl Zeichen, die ausgelesen werden sollen, **Delim** = Ende-Zeichen (Delimiter) bis zu dem ausgelesen werden soll:

10 print mpeek(@A\$)	es werden max. 256 Bytes ausgelesen, das Null-Byte wird als String-Ende erkannt.
20 let T(8)= mpeek(0x1000000, 64)	liest max. 64 Bytes aus dem CCM-RAM bis zum String-Ende
30 print mpeek(@"pahlbasic", 99, 34)	liest max. 99 Bytes bis Zeichen 34 (Anführungszeichen)

## path\$

liefert den aktuellen Verzeichnis-Pfad auf der SD-Card, z.B.: **VOLUME:\Armbasic32\**  
Seit V1.07 geht: `let A$=path$` und: `cd A$`.

Der F405RG und F407ZG kennen auch **path\$(Drive-Nummer)**, gespeicherte Pfade von Drive1 und Drive2.

## prog\$

**prog\$** liefert den Dateinamen, Dateidatum und –uhrzeit der geladenen Programmdatei, so wie sie auch beim Laden des Programms angezeigt werden.

```
load 3
print prog$          Apfelmaennchen, created on Mo, 15.01.2024, 19:45:25
```

Werden Name, Datum und Uhrzeit einzeln benötigt, kann man sie mit mpeek() holen:

```
print mpeek(sys(105)), mpeek(sys(106)), mpeek(sys(107))
```

## result\$

Ergebnis-String einer User-Stringfunktion. Siehe [f\\$](#).

## sensor\$

holt die Serien-Nummer eines Dallas-One-Wire-Temperatur-Sensors in einen *String*.

Siehe auch [sensor\(\)](#).

Sensor an PA7 anschliessen (Anschlussbild siehe [Anhang](#)):

```
> print sensor$

Temperature Sensor is DS18B20           zusätzliche System-Meldung
6630167858422087464
ready
```

## stamp\$

`stamp$` oder: `stamp$(Zeitwert)`

stellt Zeitstempel als String dar:

`stamp$` liefert einen String, der den mit [stamp](#) gespeicherten (System)-Zeit-  
<https://www.muezech.de/> pahlbasic <https://github.com/sitosan/pahlbasic>  
130

stempel als Zeichenkette darstellt.

### stamp\$(Zeitwert)

liefert einen String, der den aus **Zeitwert** gebildeten Zeitstempel als Zeichenkette darstellt. Zeitwert muss in einer Variablen (**A-Z**) gespeichert sein.

**Zeitwert** ist der Inhalt des RTC-Time-Registers plus der Inhalt des RTC-Date-Registers. Man könnte die Werte mit peek() erhalten. Es geht aber einfacher: **stamp** generiert sie:

<b>ex.:</b>	
10 stamp	erzeugt System-Zeitstempel
20 print stamp\$	Ergebnis: Mo, 23.10.2023, 11:02:34

und speichert sie auch:

<b>ex.:</b>	
10 stamp A	erzeugt und speichert den System-Stempel
20 print stamp\$(A)	Ergebnis: Mo, 23.10.2023, 11:03:55

### str\$()

#### str\$(Ausdruck [, Breite ] )

verwandelt das Ergebnis eines **Rechenausdrucks** in einen **String**.

<b>ex.:</b>	
5 input A	
10 let A\$ = str\$(A)	verwandelt den Wert der Variablen A in einen String.
20 print A\$(1, 2)	den String kann man zerlegen
30 let D\$ = str\$(4+(3*B), 12)	Rechenergebnis in einen String mit Breite 12.
40 print D\$	Ausdruck des Ergebnisses

Soll der erzeugte String eine bestimmte (Mindest-) **Breite** haben, kann man diese zusätzlich angeben. **Breite** muss mindestens der zu erwartenden Stellenzahl +1 entsprechen.

Es werden zusätzliche Spaces eingefügt, bis **Breite** erreicht ist. Dies ist für den Ausdruck von unterschiedlich großen Zahlen an derselben Position (Terminal oder LCD) interessant, damit Ziffern von vorherigen Ausdrücken nicht 'stehen bleiben', sondern überschrieben werden:

<b>ex.:</b>	
10 let N = 46	
20 let A\$ = str\$(N, 5)	A\$ ist immer 5 Zeichen breit, auch wenn N z.B. nur 2 Stellen hat.

Es kommt zwar nicht oft vor im Controller-Alltag aber trotzdem: für Ausgaben bei denen die Zahlen  $> 10^7$  oder  $< 10^{-4}$  werden können, gibt es:

### str\$(Ausdruck, sci)

nehmen die Zahlen sehr große oder sehr kleine Werte an, wird die wissenschaftliche Darstellung gewählt. "Dazwischen" normale Darstellung mit bis zu 15 Nachkommastellen.

```
string()
```

```
    string(Anzahl, Zeichen)
```

string erzeugt eine Zeichenkette, die aus der **Anzahl** gleicher **Zeichen** besteht.

**ex.:**

```
10 print string(15, "+")           Ergebnis: ++++++++  
20 print string(2*N+1, 32+C)
```

ucase

```
ucase String
```

der (**Teil-)**Ergebnis-String wird in Großbuchstaben verwandelt. ucase ist ein String-operator.

**ex.:**

```
10 let A$ = "hallo welt"  
20 print ucase A$(7, 4)           Ergebnis: WELT
```

## Vergleiche

=, <>, >, <, in

Strings können auf Gleichheit „=“ oder Ungleichheit „<>“ geprüft werden oder ob der erste im zweiten String enthalten (in) ist:

```
ex.:
10 if (A$=B$) then print "Die Namen sind gleich"

20 print (B$ <> "Montag")

30 if ("pahlbasic"=C$) : ...                                Zeichenkette mit Stringvariable

40 if (date=="Mo, 01.12.2011") : ...                        Systemvariable mit Zeichenkette

50 let A = ("12:24:30"=clock)                                Ergebnis des Vergleichs in Variable

60 if ("pahlbasic" in T(0)) then print "gefunden"           Zeichenkette in Textarray-String suchen
```

Bitte die Klammern beachten!

Strings können seit V1.05A26 auch auf grösser ">" oder kleiner "<" (lexikalisch) geprüft werden:

```
10 print ("Montag" > "Dienstag")                            (ist wahr)
20 print ("Montag" > "Mont")                                 (ist wahr)
30 print ("Montabaur" > "Montag")                           (ist falsch)
```

(A\$>B\$):                    A\$ steht im Lexikon hinter B\$  
(A\$<B\$):                    A\$ steht im Lexikon vor B\$

Ist die lexikalische Position gleich, wird die String-Länge zum Vergleich herangezogen.

## Einzelnes oder mehrere Zeichen (Teilketten)

einzelnes Zeichen:

**Stringvariable(Zeichenposition)**

Eingefasst von Klammern werden einzelne Zeichen als Zahlen (Bytes) interpretiert. Auch das Ergebnis von Stringvergleichen ist numerisch – also müssen Klammern gesetzt werden:

**ex.:**

10 let C\$(3) = "E"	Tauscht das 3. Zeichen in der Stringvariablen C\$ gegen „E“ aus. (Voraussetzung: C\$ ist nicht leer) links vom '=' ein Zeichen, rechts auch ein Zeichen, keine Klammer
10 let B\$ = "1234567890" 20 let A = (B\$(5)) 30 print A	weist der Variablen A den Ascii-Wert des 5. Zeichens von B\$ zu links eine Zahl also auch rechts eine Zahl (Klammer nötig)
70 if (B\$(5) = "A") : ...	und so prüft man auf ein Zeichen an bestimmter Position (String-Vergleich in Klammern)

oder:

70 if (B\$(5) = 65) : ...      das gleiche numerisch (Zahlenvergleich)

Zeichenposition darf auch ein komplexer Rechenausdruck sein.

## Position Null

X\$(0) ist ein virtuelles Zeichen, es dient dazu, Zeichen an das Ende der Zeichenkette anzuhängen. (X\$ steht für: A\$ ... H\$)

let A\$(0) = "d"      hängt das Zeichen „d“ an das Ende von A\$.

Das funktioniert auch mit leeren Zeichenketten:

**ex.:**  
10 for N=1 to 10 : get A : let A\$(0) = A : next N

Liest 10 Zeichen vom Terminal ein und bildet den A\$ daraus.

Das Textarray funktioniert analog:

let T(9,0) = "f"      hängt das Zeichen „f“ an das Ende von T(9).

## Teilzeichenkette

**Stringvariable(Position, Anzahl)**

greift auf einen Teilstring innerhalb eines Strings zu. Man kann ihn lesen oder verändern. In die „Mitte“ einer *leeren* Zeichenkette kann man *nicht* schreiben:

let A\$ = "12345678901234"	der ganze String
print A\$(4, 3)	Ausdruck Teilstring: 456
let A\$(4, 8) = clock	
print A\$	Ergebnis: 12314:38:06234

## Textarray

Auch das Textarray lässt den Zugriff auf einzelne Zeichen oder Teilzeichenketten zu. Die Anwendung ist analog zu den Strings. Zusätzliche Angabe der Array-Element-Nr. (**String-Nr.**) erforderlich:

**T(String-Nr., Startposition, Anzahl)**

Teilzeichenkette mit Anzahl Zeichen ab Startzeichen:

**ex.:**  
10 if (T(8,1,14)=date) goto 200

**T(String-Nr., Zeichen-Nr.)**

Einzelnes Zeichen:

**ex.:**  
10 print T(120, 19) 19. Zeichen in String Nr. 120

## Negative Werte

für Startposition zählen vom *rechten* String-Ende. Gilt für Array- und „normale“ Strings.

Ersatz für **right\$** und **left\$** und **mid\$** in anderen Basic-Dialekt:

Fremder Dialekt:

```
10 print right$(A$, 6)
20 print left$(A$, 7)
30 print mid$(A$, 2, 4)
```

Ab dem 5. Zeichen bis zum Ende des Strings:

pahlbasic:

```
10 print A$(-6, 6)
20 print A$(1, 7)
30 print A$(2, 4)
40 print A$(5, len(A$))
```

Fehlt nur noch zu sagen: Funktioniert auch mit dem Textarray.

Siehe auch den neuen String-Operator [mid](#).

## Konstanten

true

intern verwendet für das Ergebnis eines Vergleichs.  
`true = 4294967295`

binär: `%11111111111111111111111111111111`  
(-1 als Integer in anderen Basic-Dialekten)

```
print 4>3          Ergebnis: 4294967295
```

false

intern verwendet für das Ergebnis eines Vergleichs.  
`false = 0`

```
print 3>4          Ergebnis: 0
```

pi

die Kreis-Zahl  
`pi = 3.1415926`

e

die Eulersche Zahl  
`e = 2.7182818`

`true`, `false`, `pi` und `e` sind Schlüsselwörter und können auch im Basic-Programm verwendet werden.

## Zahlen

außer dezimalen Fließkommazahlen kennt pahlbasic auch Hexadezimalzahlen und Binärzahlen. Intern werden aber alle Zahlen als 64-Bit Floating-Point-Zahlen verarbeitet.

### Hexadezimalzahlen

werden durch ein vorangestelltes „**0x**“ gekennzeichnet.

**z.B.:**

**0xF00C**

(dez. 61452)

max. 8 Stellen

**ex.:**

```
10 let relay = 0xAaaa
```

**&h** versteht pahlbasic aber auch und übersetzt es in **0x**.

### Binärzahlen

werden durch ein vorangestelltes „**%**“ gekennzeichnet.

**z.B.:**

**%10010011**

(dez. 147)

max. 32 Stellen

**ex.:**

```
10 let relay = %1010101010101010
```

Bit-Nr. 0 ist ganz rechts

**0b** und **&b** versteht pahlbasic aber auch und übersetzt es in **%**.

Die verschiedenen Zahlentypen können in einem Rechenausdruck gemischt werden. Falsche Kennzeichnung führt zu Programm-Abbruch mit [Fehlermeldung](#).

[www.pahlbasic.de](http://www.pahlbasic.de)

## Fehlermeldungen

- 1 **illegal direct**..... nicht im Direktmodus (`if`, `return`, `on...`)
- 2 **syntax error**..... eine Anweisung ist falsch (geschrieben). Bei der Eingabe gibt der Interpreter einen Hinweis auf die Position des Fehlers in der Form:  
.....**X** wobei **X** die Fehlerstelle markiert. (wie MCS51 Basic), sonst zur Laufzeit.
- 3 **expected ", () or #**..... Klammer, #, Komma, "" fehlt oder ist nicht geschlossen
- 4 **expected ' to '**..... `to` oder Endwert fehlt in `for`-Schleife
- 5 **next without for**..... ein `for` fehlt
- 6 **return without gosub**..... ein `gosub` weniger als `returns`
- 7 **expected '='**..... = Zeichen in `let`, `for` oder `config` fehlt.
- 8 **line number?**..... Zeilen-Nummer fehlt, ist Null oder existiert nicht.
- 9 **stack error**..... `break` wurde ausgeführt, obwohl keine Adresse auf dem Stapel war, Rekursion zu groß > 10 etc.
- 10 **dcf77 or gps ctrl**..... System-Uhr ist synchronisiert, Zeit-einstellung nicht mehr nötig
- 11 **not in prog mode**..... Nur im Direktmodus erlaubt. (`new`, `run`, `list`, `save`, `dir`, `del`)
- 12 **can't continue**..... Nach einem Programmfehler kann nicht mit `cont` fortgesetzt werden.
- 13 **bad argument**..... Falsches Argument oder Division durch 0 oder Wurzel aus negativer Zahl
- 14 **out of memory**..... zu viele Klammern, `for/next`-, `do/loop`-Schleifen, `gosub` (max. 20 Ebenen) oder Programmspeicher voll.
- 15 **loop without do**..... ein `do` fehlt (auch `wend` ohne `while`)
- 16 **no program**..... kein Programm im RAM-Speicher bei `save` / Datei ist kein Programm
- 17 **no ext. eeprom**..... kein externer EEPROM- / Flash-Speicher vorhanden.
- 18 **file not found**..... Datei / Programm nicht gefunden
- 19 **hardware**..... Die Hardware ist für die Anweisung nicht geeignet / eingerichtet / gestartet oder SD-Card Lese-/Schreibfehler
- 20 **file too big**..... Das Programm im RAM passt nicht ins externe Flash. (bei `save`) bzw. die Dateilänge überschreitet 64KB
- 21 **Index!**..... Der Index ist zu groß (zu klein) z.B. bei Bits größer als 31
- 22 **adress error**..... User-Funktionen müssen vor ihrer

Verwendung angemeldet werden.

- 23 **file already exists**..... Datei(-Namen) gibt es schon
- 24 **Data Type mismatch**..... numerischer Wert statt String oder umgekehrt
- 25 **file not open**..... vor dem Lesen oder Schreiben muss die Datei geöffnet werden
- 26 **no read file**..... Datei ist nicht zum Lesen geöffnet
- 27 **no write file**..... Datei ist nicht zum Schreiben geöffnet
- 28 **file end reached**..... Datei-Ende erreicht
- 29 **no Data File**..... es handelt sich *nicht* um eine sequentielle Daten-Datei
- 30 **filehandler??**..... der Filehandler wird bereits benutzt / die Nummer ist falsch
- 31 **soft break**..... Fehler für Zwangsabbruch (für User)
- 32 **user break**..... Das Programm wurde vom Nutzer mit **Strg+c** beendet, cont ist möglich

Ab Version 1.05 gibt es zusätzliche Hinweise zur Fehlerbeschreibung wie z.B.: "[max. Klammerebene überschritten](#)"

Fehler, die in User-Prozeduren oder -Funktionen auftreten, werden vom Hauptprogramm in der Zeile gemeldet, die die Prozedur oder Funktion gestartet hat. Damit man den Fehler trotzdem findet, geben die aufgerufenen Programm-Instanzen zusätzliche Meldungen aus. Die erste Meldung ist die wichtigste.

Es werden die meisten Fehler erkannt, jedoch kann man nicht alle Fehler vorhersehen.

Hauptaugenmerk liegt darauf, das Basic absturzsicher zu machen. Commodore Basic konnte man zum Absturz bringen, pahlbasic sicher auch. Dann hilft nur noch der Resetknopf.

Ebenfalls hilft nur der Resetknopf bei Exception-Fehlern wie z.B. 'bus-error' oder 'memory-error'. Hinweise für einen Bug (der Prozessor soll auf ein Objekt/Speicherbereich zugreifen, das/der nicht definiert ist). Solche Fehler bitte unbedingt melden, am besten mit einem Beispiel, das den Fehler provoziert.

## Unterschiede zu anderen Tiny Basic-Dialekten

pahlbasic hält sich meistens an Tiny-Basic Vorbilder wie MCS51 Basic. Aber bedingt durch die unterschiedliche Hardware, wurden bewusst Änderungen vorgenommen. Einer der wichtigsten Gründe für Abweichungen war der kleine zur Verfügung stehende RAM-Speicher, da pahlbasic schon auf einem AVR-Controller (ATMEG644) ohne externen RAM-Speicher lief.

**Besonderheiten sind:**

Die **vordefinierten Variablen A-Z**, die Arrays **A()** ...**J()** sowie die Stringvariablen **A\$...H\$** und das Text-Array **T()** sind schon vor ihrer Verwendung vorhanden. (kein dim oder declare nötig). Sie sind die Speicher-Elemente mit dem schnellsten Zugriff in pahlbasic. **clear-/run-/new** setzen lediglich ihren *Inhalt* auf Null bzw. Nullstring.

**Variablen mit Namen** werden angelegt, sobald sie eine Wert-Zuweisungen erhalten. Die Zugriffszeit wächst mit der Namenslänge und der Anzahl der Variablen. Sie *existieren* bis zur nächsten **clear-/run-/new**-Anweisung (entsprechen üblichen Interpreter-Variablen).

**Lokale Variablen A-J** (und jetzt auch die Stringvariablen **A\$** und **B\$**) in Prozeduren und Funktionen werden bei Start der Prozedur / Funktion angelegt und verhalten sich danach wie die vordefinierten Variablen. Sie verschwinden mit Beendigung der Prozedur / Funktion.

Intern wird für numerische Variablen die IEE754 Fliesskomma-Darstellung mit 64 Bit verwendet, das würde für 14-16 Dezimalstellen reichen. Die Ein-/Ausgaberoutinen stammen jedoch noch aus Version 1.03 (32-Bit), die Standart-Printausgabe erreicht daher nur etwa 6-8 Stellen Genauigkeit. Große Zahlen werden (Workaround) als 64-Bit Integers ausgegeben – z.B. 32-Bit-Adressen. (Stand Okt. 2021)

Wenn die zu erwartenden Zahlen Werte > 10 000 000 oder < 0.0001 annehmen können (kommt im Controller-Alltag eher selten vor), schafft die Funktion **str\$(Zahl, sci)** Abhilfe. (hinzugefügt Mai 2022). Als Standart waren mir Zahlen mit bis zu 16 Stellen oder wissenschaftlicher Darstellung dann aber doch zu unhandlich.

**Vergleiche:** **wahr** ist in pahlbasic alles, was nicht **Null** ist. (machen einige andere Basic-Dialekte auch so)

10 **while** 0.3 : **wend** ewige Schleife – also ist **0.3** wahr

**Bits:** Zuweisung eines Werts an ein Bit – der Interpreter überprüft, ob das Bit **0** im Wert gesetzt (also der Wert ungerade) ist, dann wird auch das entsprechende Bit gesetzt. **let A.5 = 2** das Bit wird nicht gesetzt **let A.5 = 3** das Bit wird gesetzt. Bleibt man bei **0** und **1** ist es aber auch gut. ☺

**Rangordnung:** Operatoren haben keine Rangordnung. Die Programmiersprache C hat 15 Ebenen. Das merkt sich keiner. Für Klarheit setzt eh jeder Klammern. Habe ich erst gar nicht versucht, einzubauen, verlangsamt die Rechenroutine auch nur unnötig.

**Arrays:** sind stets 0-basiert, das gilt auch für das Textarray. Stringelemente (Zeichen) beginnen jedoch aus Kompatibilitätsgründen mit der Nummer 1.

**Relais, Eingänge:** starten mit der Nummer **1** (betrifft die Anweisungen **impuls**, **on**, **reset**, **spi**, **set**, **toggle** und **wait**), die korrespondierenden Bits der zugehörigen Systemvariablen (**relay**, **diginp**) aber mit Nummer **0** (betrifft alle Rechenausdrücke). Im Text wird darauf hingewiesen. Relais 1 hängt eben an Bit 0 von Port D (identisch mit Systemvariable **relay**). Gilt nur für die F405/7-Boards.

## Beispiele für Syntax-Abweichungen:

Bei der Syntax gibt es nur wenige Abweichungen zu anderen Vintage-Basic-Dialektken, sodass mit wenigen Änderungen viele alte Basic-Programme laufen sollten:

**Bezeichner** Variablen-Namen: numerische Variablen werden durch '**\$**' gekennzeichnet, Stringvariablen haben keinen Namen. Texte dienen in pahlbasic lediglich als Hinweise und Ausgaben.

**instr** die Funktion **instr** heißt in pahlbasic: **in**: let X = (**A\$ in B\$**)

**def fnX()** heißt in pahlbasic: **deffn Name() as n**

**procedure, function, Label oder auch sub procedure, sub function**

heißen in pahlbasic **sub**, **function** und **proc** und arbeiten so ähnlich. In einigen Basic-Dialekten ruft man Prozeduren mit **call** auf, in anderen genügt der Name der Prozedur. In pahlbasic kann man mit **call** vollständig gekapselte Prozeduren aufrufen. Vor Funktionsnamen muss **fn** für numerische Funktionen und **f\$** für Stringfunktionen stehen.

**do ...until** Syntax weicht vom ANSI-Standard ab. (ANSI: do while ...loop, do until...loop, do...loop while, do...loop until). Mit **do...until** und **while...wend** sowie **for...next** lassen sich aber sämtliche Schleifenprobleme lösen.

**let** ist in den meisten Basic-Dialekten optional und wird meist weggelassen – nicht in pahlbasic.

**mid\$** gibt es in pahlbasic nicht. Ebenso wenig, wie **left\$** oder **right\$**.

Die Array-Schreibweise in pahlbasic (in Anlehnung an ANSI) macht Extra-String-Funktionen für den Zugriff auf Zeichen/Teilketten so gut wie überflüssig. Damit man aber auch Strings wie **date**, **clock** oder **gps\$** beschneiden kann, gibt es den String-Operator **mid**.

**open / close**

z.Z. nur für sequentielle Dateien verfügbar. Access mode OUTIN ist nicht vorgesehen, aber man kann Dateien gleichzeitig als Output und Input (Flash-Chip) öffnen. Beliebig überschreibbare Dateien kommen mit den EEPROMS und mit der SD-Card in einer zukünftigen Version.

**lineinput#** entspricht **input #**  
**input#** entspricht **read #**

**select / case**

die Anweisung **select case** heisst in pahlbasic **select**, die Anweisung **case else** heisst in pahlbasic **celse**, die Anweisung **end select** oder **select end** heisst in pahlbasic **selend**. pahlbasic ist für schreibfaule ideal. Siehe auch die neuen Abkürzungen.

# Neu in pahlbasic

Seit kurzem sind die Beta-Versionen online. Anregungen werden in der nächsten Zeit umgesetzt (sofern sie ins Konzept passen).

Die gemeldeten Bugs werden protokolliert und beseitigt. ☺

## Hier werden die aktuellen Änderungen aufgeführt:

### Bis V1.14:

Modbus-Anweisungen hinzugefügt. Anweisung `server` hinzugefügt (verwandelt den Basic-Computer in ein 'Modbus'-Modul – mit eigenwilligem 'Modbus'-Protokoll).

`lcall` ruft Prozeduren im RAM auf, die `lcall` selbst von SD-Card nachgeladen hat.

`rcall` kann jetzt auch Prozeduren vom ROM aus im RAM starten. (siehe `call`)

`rload` lädt Basic-Programm-Dateien von SD-Card direkt ins ROM (siehe `load`).

Der ROM-Speicher für BASIC-Programme kann jetzt vergrößert werden, wenn man den Sektor 10 ('Experimentier-Sektor') nicht für MaschinenSprache-Programme oder Pokes braucht. Die Einstellung `config start r=1` verdoppelt den ROM-Speicher von 128K auf 256K. So grosse Programme bekommt man nur mit `rload` ins ROM.

Anweisung `wdog` bringt einen Watchdog für das Basic.

TFT-Zeichensätze teilvollständigt (Codes 160...255). Der Zeichensatz Impact wurde etwas vergrößert: 53x35 (Impact aber nur bis Code 126).

Support für CCS881 und die Anweisung `ccs` wurden entfernt.

Anweisung `get dir$` hinzugefügt. Ermöglicht den Zugriff auf SD-Card Directory-Einträge. Siehe auch Beispieldatei `Player.PBAS`.

Das Paragraph-Zeichen '§' liegt jetzt zusätzlich auf Funktionstaste 9 – nicht jeder hat ein deutsches Keyboard (kann überschrieben werden mit `fkey`).

### V1.15

#### Januar '25:

Im REPL-Modus (Erweiterung des Kommando-Modus) können jetzt mehrere Anweisungen (durch Doppelpunkt getrennt) eingetippt und ausgeführt werden. Die Anzahl der Anweisungen ist durch die Zeilenlänge beschränkt (123 Zeichen). Dieser REPL-Modus wird durch den Doppelpunkt (oder `then`) aktiviert. Die eingetippten Kommandos werden wie ein kleines Mini-Programm nacheinander ausgeführt. Es lassen sich auch Anweisungen ausführen, die eigentlich nur im Programm-Modus funktionieren (z.B. Loops oder `if then`). Reine Kommandos (`run`, `list`,...) können weiterhin nur im Kommando-Modus (1 Anweisung, ohne Doppelpunkt) gestartet werden – den REPL-Modus gibt's nur für F405/F407.

Stringverkettung hat jetzt den Operator '\n' bekommen (fügt Newline ein). Siehe `print`.

Poke kann jetzt auch 32Bit Floats.

#### geplant für 2025:

F405/7:           NEO-Pixel-LEDs with 32Bit colors  
                  Text auf NEO-Pixel-Panel 8x32  
                  playing calculated and sampled sound files with connected MIDI  
                  Keyboard / from MIDI files  
                  Support: 3 Channel PWM-Modbus Modul  
                  Small Database on SD-Card

Display-Controller: E-Paper mit SSD1683 (400x300 Pixel)

Boards:           \* STM32F405RG (64Pins, 'Pyboard')

F407ZG-Board:    Soundkarte mit VS1003/VS1053  
                  \* ST7793 (Arduino)  
                  Abspielen von MP3-, Ogg-Vorbis-, und MIDI-Dateien

✓ = bereits erledigt   \* = priorisiert

## Anhang 1

Toucheingabe mit der keypad-Anweisung:



Ist die Variable numerisch sind nur Zahleneingaben möglich. Wird ein String erwartet, sind alle Tasten aktiv. Gross-/Kleinschreibung erfolgt durch kurzen oder langen Klick, z.Z. gibt es noch keine Zeichen für Escape, Return und Rückschritt. Es sollte ein Stift für die Eingabe benutzt werden. Das Fragezeichen an erster Eingabeposition schaltet den Formeleingabemodus ein (Taschenrechner).

4-Buttons Dialogbox erzeugt mit: draw „br“,A:



OK-Messagebox  
erzeugt mit:  
let H\$ = „bitte bestaetigen“ : draw „m“

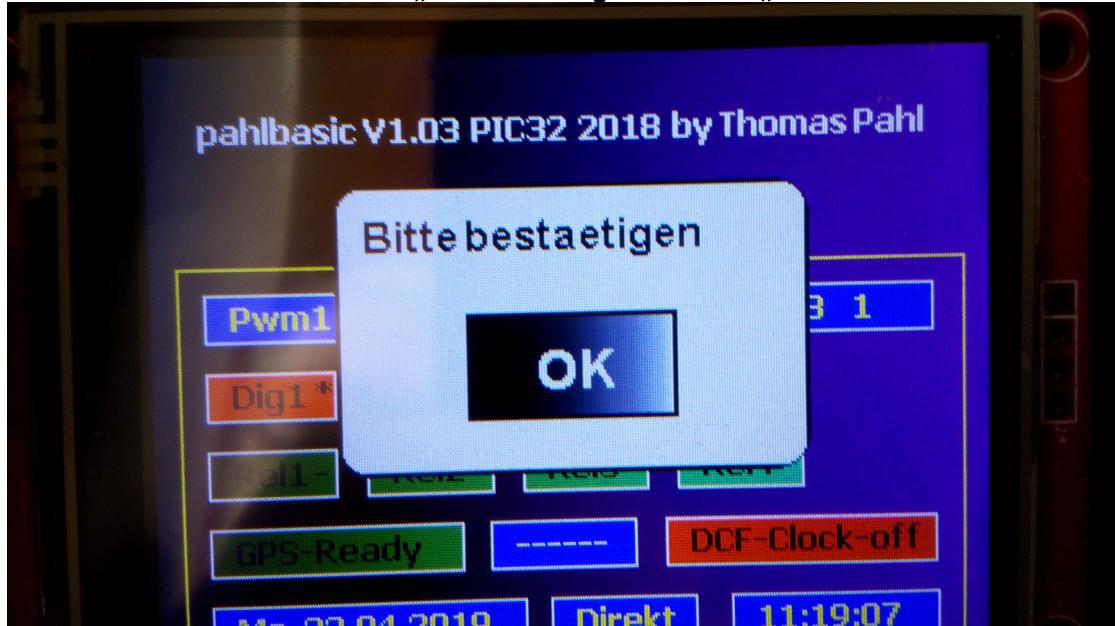


Abb.7 Messagebox auf dem TFT-LCD

Der Info-Screen im Direktmodus

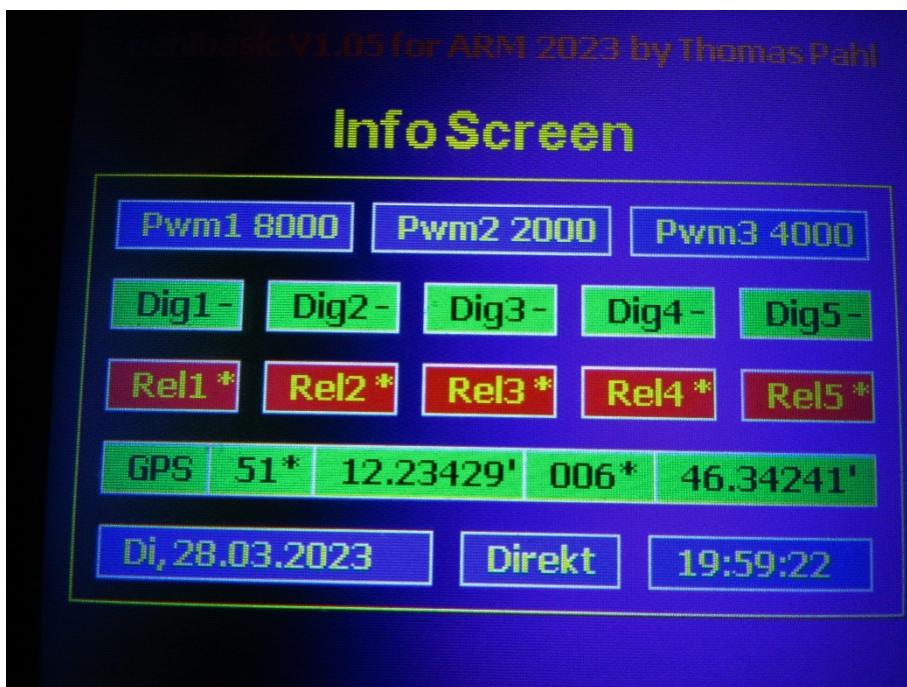


Abb.8 Info-Screen auf dem TFT-LCD

Statusanzeigen von PWMs,

Digitalen Inputs,

Outputs,

GPS

und der Uhr

Das TFT-Display ist nur 3,5 Zoll groß!

## Die Funktion sys()

### Die Funktion sys() im Detail:

- sys(0): Task-Nr. Vom System vergebene Nummer für die aktuell ausgeführte Prozedur- oder Funktions-Ebene. Das Hauptprogramm hat stets die Nr.1, der Direkt-Modus die Nr. 0.
- sys(1): gosub-Ebenen-Zähler. Bei jedem **gosub** (auch **call**, **do** oder **for**) legt der Interpreter die Adresse für den Rücksprung mit **return** (auch **while**, **until** oder **next**) auf einen Stapel und erhöht den Zähler um 1.
- sys(2): Rechenstapel-Ebenen-Zähler. Bei jeder geöffneten Klammer legt der Interpreter die Rechen-Zwischen-Ergebnisse auf einen Stapel und erhöht den Zähler um 1. Ebenso bei jeder Index- oder Bit-Berechnung, Prozedur oder Funktion. Wird die Klammer geschlossen, die Indexberechnung oder Funktion beendet, geht es wieder eine Stufe tiefer.
- sys(3): Die Frequenz des Signals, das am Digitaleingang 6 (diginp 6) anliegt. Der Eingang muss zuvor mit **set diginp 6 as fcount** umgeschaltet worden sein. Die Torzeit beträgt 1 Sekunde, sodass der Wert in Hz gemessen wird. Die Messung erfolgt fortlaufend.
- sys(4): Anzahl Impulse des Signals, das am Digitaleingang 6 (diginp 6) anliegt. Der Eingang muss zuvor mit **set diginp 6 as icount** (1 oder 2) umgeschaltet worden sein. Es werden alle Impulse seit der letzten sys(4)-Abfrage gezählt, da der Zähler nach der Abfrage auf Null gesetzt wird. Der Zählzeitraum ist beliebig. Es wird der Zähler 9 verwendet (16Bit mit einer Software-Erweiterung auf 32Bit) sys(4) liest diese Zählerkombi direkt aus.
- sys(5): Printposition auf dem TFT-LCD. Der 32Bit-Wert enthält Zeile und Spalte auf denen die nächste Ausgabe erfolgt. Untere 16-Bit: Zeile, obere 16-Bit: Spalte. Mit **low()** und **high()** auflösen.
- sys(6): Cursorposition auf dem Terminal (Konsole). Der 32Bit-Wert enthält Zeile und Spalte der aktuellen Cursorposition. Untere 16-Bit: Zeile, obere 16-Bit: Spalte. Mit **low()** und **high()** auflösen. Es wird eine VT100-Sequenz an das Terminal gesendet und das Terminal antwortet ebenfalls mit einer Sequenz, die die gewünschten Positionen enthält.
- sys(7): Kalibrier-Wert der Echtzeit-Uhr (RTC). Der ppm-Wert entspricht dem Wert, der zur Korrektur mit **set clock** eingegeben wurde.
- sys(8): Sommerzeit=true, Winterzeit=0. Wird vom Interpreter aus dem aktuellen Datum berechnet.
- sys(9) Zuletzt aufgetretener I2C1 oder I2C2-Fehler. 1=Read, 2=Write, 3=Idle, 4=Start
- sys(10) Inhalt des RTC\_CR-Registers
- sys(11) Inhalt des RTC\_BKP5R-Registers, siehe dump8: verschiedene Einstellungen z.B. Bit 5 = Insert-Mode on/off, Bit 6 = I2C-Scanning on/off
- sys(12) Inhalt des RTC\_BKP9R-Registers, siehe dump8: erkannte I2C-Sensoren z.B. Bit 0 = HTU31 präsent: ja/nein, Bit 1 = BME280 präsent: ja/nein
- sys(13) Inhalt des EXTI\_IMR-Registers (aktive externe on-Interrupts: diginp, touch etc. – 1Bit je Eingang, Bit 0=diginp1, Bit15=touch)
- sys(14) Auflösung (max. Wert) der PWMs (frequenzabhängig), z.B.: pwm1 auf 15% setzen:  
**let pwm1 = sys(14) \* 15 / 100**

sys(15)	Zustand der USB-Device-Verbindung: z.B.: 0=nicht gestartet, 32=verbunden, 64=unterbrochen
sys(16)	aktuelle Prozedur- oder Funktions-Nummer (as-Nummer) Prozeduren 1...32, Funktionen 33...37, ohne=38
sys(17)	loop-Zähler zählt Schleifen-Verschachtelungstiefe
sys(18)	Anzahl WS2812B-LEDs (wie mit config eingestellt)
sys(19)	Adresse des LED-Farb-Speichers (256 x 32-Bit)
sys(20)	PWM-Frequenz
sys(21)	Anzahl der Zeichen im Empfangspuffer der seriellen Schnittstelle 1
sys(22)	Anzahl der Zeichen im Empfangspuffer der seriellen Schnittstelle 2 (MIDI)
sys(23)	Anzahl der Zeichen im Empfangspuffer der seriellen Schnittstelle 3 (GPS)
sys(24)	Anzahl der Zeichen im Empfangspuffer der seriellen Schnittstelle 4
sys(25)	Anzahl der Zeichen im Empfangspuffer der USB-Schnittstelle (Device & Host)
sys(26)	aktuelle Ausgabe-Schnittstelle
sys(27)	aktuelle Programmzeilen-Nummer
sys(28)	GPS Positionsdaten valid=true, not valid=false
sys(29)	Schnittstellen-Nummer der Konsole
sys(30)	Zustand der Relais-Ausgänge nach Reset, Bits 0...15, 1=Ein, 0=Aus (analog zur Systemvariable <a href="#">relay</a> )
sys(31)	Zustand der Relais-Ausgänge nach Reset, Bits 0...15, 1=Opendrain, 0=Komplementär (analog zur Systemvariable <a href="#">relay</a> )
sys(32)	Zustand der Pullup-/Pulldown-Widerstände der digitalen Eingänge, Bits 0...11, 1=Pullup, 0=Pulldown (analog zur Systemvariable <a href="#">diginp</a> )
sys(33)	Zustand der Entprellung der digitalen Eingänge, Bits 0...4, eine 1 bedeutet Totzeit-Timer eingeschaltet, 0=ausgeschaltet (analog zur Systemvariable <a href="#">diginp</a> , aber nur Bit 0...4)
sys(34):	Länge des Impuls-Signals, das am Digitaleingang 6 ( <a href="#">diginp 6</a> ) anliegt. Der Eingang muss zuvor mit <a href="#">set diginp 6 as icount2</a> umgeschaltet worden sein. sys(34) wartet auf den Impuls und dann auf sein Ende. Danach wird der Zähler ausgelesen. Der Zähler ist mit 1MHz getaktet – misst also µs, der Pegel an <a href="#">diginp 6</a> ist in diesem Fall das Tor.
sys(35):	Farbverlaufsrichtung 0=von oben nach unten, 1=von links nach rechts
sys(36):	Farbverlaufs-Startfarbe
sys(37):	Farbverlaufs-Endfarbe
sys(38)	USB-Host gestartet? (0=nicht gestartet, 1=gestartet)

sys(39)	USB-Device gestartet? (0=nicht gestartet, 1=gestartet)
sys(40)	Adresse der aktuellen Prozedur oder Funktion
sys(41)	Geschwindigkeit der seriellen Schnittstelle 1 (Baud)
sys(42)	Geschwindigkeit der seriellen Schnittstelle 2 (Baud)
sys(43)	Geschwindigkeit der seriellen Schnittstelle 3 (Baud)
sys(44)	Geschwindigkeit der seriellen Schnittstelle 4 (Baud)
sys(45)	Nummer der letzten gedrückten Taste auf dem USB-Keyboard
sys(46)	Nummer der letzten losgelassenen Taste auf dem USB-Keyboard (wird von der Input-Routine gelöscht)
sys(47)	nächster freier Filehandler für sequentielle Dateien (nur auf dem Flash Memory)
sys(48)	Anzahl Textzeilen auf dem LCD
sys(49)	Anzahl Textspalten auf dem LCD
sys(50)	Attack-Faktor – für interne Berechnung.
sys(51)	Decay-Faktor – für interne Berechnung.
sys(52)	Release-Faktor – für interne Berechnung.
sys(53)	Display-Typ-Nummer. Siehe <a href="#">config lcd</a> .
sys(54)	Farbpalette eingeschaltet? ein=true, aus=0
sys(55)	Zeichenfarbe (Farbwert, wenn Palette=aus - Farbnummer, wenn Palette=ein)
sys(56)	Hintergrundfarbe (Farbwert, wenn Palette=aus - Farbnummer, wenn Palette=ein)
sys(57)	Füllfarbe (Farbwert, wenn Palette=aus - Farbnummer, wenn Palette=ein)
sys(58)	Textfarbe (Farbwert, wenn Palette=aus - Farbnummer, wenn Palette=ein)
sys(59)	Strichstärke
sys(60)	Display rotiert=true, nicht rotiert=0, (Rotation=180°)
sys(61)	Restlaufzeit des Impulstimers 1 in Millisekunden
sys(62)	Restlaufzeit des Impulstimers 2 in Millisekunden
sys(63)	Restlaufzeit des Impulstimers 3 in Millisekunden
sys(64)	Restlaufzeit des Impulstimers 4 in Millisekunden
sys(65)	Nächster freier Impulskanal ( <b>1...4</b> bzw. <b>0</b> wenn kein Kanal frei ist)
sys(66)	Protokollierter Zeitwert im RTC-Time-Register-Format (RTC = Hardware-Uhr)
sys(67)	Protokollierter Datumswert im RTC-Date-Register-Format

sys(68)	Aktueller Zeitwert im RTC-Time-Register-Format
sys(69)	Aktueller Datumswert im RTC-Date-Register-Format
sys(70 + <u>Interrupt-Nr.</u> )	Adresse der Interrupt-Routine, auch zum Prüfen, ob Interrupt aktiv ist (inaktiv -> Adresse = 0), z.B. <code>milli3 on</code> -Interrupt: <code>print sys(70 + 13)</code> siehe: vom 'Basic nutzbare Hardware-Interrupts'
sys(95)	Nummer des aktuellen Interrupts. Nur innerhalb von Interrupt-Service-Routinen abfragbar. (formals: Systemvariable intrpt)
sys(96)	freier Programm-Speicherplatz (formals: Systemvariable free)
sys(97)	maximale Programmlänge
sys(98)	Programmende (aktuelle Programmlänge -1)
sys(99)	CPU-Exceptions (nur für Entwicklung – im Normalbetrieb deaktiviert)
sys(100)	Adresse des Programmspeichers
sys(101)	Kompilierdatum der Firmware als Zahl : 150623 entspricht 15.06.2023
sys(102)	Es wurde (k)ein GPS-Signal auf com3 erkannt. True=GPS, 0=kein GPS
sys(103)	GPS-Parsing on=1, off=0
sys(105)	Adresse des Datei-Namen-Strings
sys(106)	Adresse des Datei-Datum-Strings
sys(107)	Adresse des Datei-Uhrzeit-Strings
sys(108)	Adresse des aktuellen Verzeichnis-Namens
sys(109)	Adresse des letzten GPS-Strings
sys(110)	Systemfrequenz in Hz (Prozessortakt)
sys(111)	Anzahl freier Cluster auf der SD-Card -- freie Bytes: <code>print sys(111)*sys(112)</code>
sys(112)	Anzahl Bytes per Cluster auf der SD-Card
sys(113)	letzter aufgetretener Fehler auf der SD-Card, siehe <b>SD-Card-Errors.txt</b>
sys(114)	Display-Breite in Pixel
sys(115)	Display-Höhe in Pixel
sys(116)	Anzahl ausgedruckter Zeilen auf dem TFT seit dem letzten <code>cls</code>
sys(117)	TFT-Scrollmodus: 0=TFT kann nicht scrollen, 1=Software Scroll, 2= Hardware Scroll
sys(118)	SD-Card-Drive-Nummer 1 oder 2 (2 nur auf F405RG und F407ZG Board)
sys(119)	CPU-ID
sys(120)	Grösse des Flash-ROMs in Bytes (Kbytes: <code>print sys(120) / 1024</code> )
sys(121)	diginp1-Zeichen (ASCII-Code)

sys(122)	diginp2-Zeichen (ASCII-Code)		
sys(123)	diginp3-Zeichen (ASCII-Code)		
sys(124)	diginp4-Zeichen (ASCII-Code)		
sys(125)	diginp5-Zeichen (ASCII-Code)		
sys(126)	Anzahl Modbus-Bytes		
sys(129)	Anzahl Dateislots auf dem Flash-Chip		
sys(142)	Anzahl Datenbits der seriellen Schnittstelle 2		
sys(143)	Anzahl Datenbits der seriellen Schnittstelle 3		
sys(144)	Anzahl Datenbits der seriellen Schnittstelle 4		
sys(200)	Programm-Nummer:	0...127	Flash-Chip; <b>2MByte: 0...31, 4MByte: 0...63,</b> <b>8MByte: 0...127</b>
		97...99	SD-Card (97=.INC, 98=.PBAS, 99=.PRG)
		100	ROM
		101	ROM mit Autostart
sys(242)	Parität der seriellen Schnittstelle 2		
sys(243)	Parität der seriellen Schnittstelle 3		
sys(244)	Parität der seriellen Schnittstelle 4		
sys(342)	Anzahl Stopbits der seriellen Schnittstelle 2		
sys(343)	Anzahl Stopbits der seriellen Schnittstelle 3		
sys(344)	Anzahl Stopbits der seriellen Schnittstelle 4		

### Der Einzelschritt-Modus:

Den Einzelschritt-Modus mit **ctrl+d'** ein-/ausschalten. Das wird mit: **d+** (ein) oder **d-** (aus) quittiert. Danach das Programm mit **run** oder **rrun** starten.

Die erste Programmzeile wird gelistet und wartet auf die Betätigung der Leertaste. Nach Betätigen der Leertaste (Space) wird die Programmzeile abgearbeitet.

Es gibt noch weitere Tasten, die in diesem Modus betätigt werden können:

'a' ...'z'	Der Wert der numerischen Variablen ( <b>A...Z</b> ) wird angezeigt.
'A' ...'J'	Das Array ( <b>A...J</b> ) wird komplett angezeigt.
§	Die Namens-Variablen werden angezeigt.
\$	Die String-Variablen werden angezeigt.

Für weitergehende Eingriffe in den Programmablauf muss das Programm mit **ctrl+c'** abgebrochen werden. Danach können alle Kommandos ausgeführt werden: z.B. Variablenwerte ändern und anschliessend das Programm mit **cont** fortsetzen.

## Die Eingabezeile

Der einfache Zeilen-Editor hat im Vergleich zur Version im ATMEGA einige Erweiterungen bekommen:

1. Einfüge-Modus hinzugefügt (In welchem Modus der Zeileneditor startet, ist konfigurierbar)
2. Entfernen-Taste
3. Cursor-links und -rechts
4. die Tasten Pos1 und Ende
5. Ctrl + e : schaltet zwischen Einfüge- und Überschreibmodus um (sowohl hin als auch her)
6. Ctrl + l : löscht die Eingabezeile
7. Ctrl + p : holt die letzte Eingabe oder gelistete Programmzeile wieder in die Eingabe
8. Ctrl + x : stellt das letzte überschriebene Zeichen wieder her (im Einfügemodus)

Der Überschreibmodus wurde von der ATMEGA-Version übernommen und bietet zum Editieren jetzt zusätzlich zum Rückschritt auch: Entfernen, Pos1- und Ende-Taste, Ctrl+l, Ctrl+p sowie Cursor links und rechts. Der Kommando-Prompt ist Lila, die Schrift ist weiss. Die Verwendung der Entfernen-Taste setzt ein VT100-Terminal voraus. Der Start-Modus wird mit **config input** festgelegt.

Der Einfüge-Modus bietet die gleichen Tastenfunktionen und Ctrl+x. Der Kommando-Prompt ist helltürkis. Die Schrift ist gelb.

Ist die Eingabelänge beschränkt, schaltet die Eingabe bei *Erreichen* der Eingabelänge auf Überschreibmodus um. Leider ist dieses Umschalten nicht zu erkennen, weil VT100 die Umschaltung des Cursors nicht zulässt **\*)**. Die Eingabezeile ändert deshalb die Eingabefarbe des Folgezeichens (auf grün). Wurde ein Zeichen fälschlich überschrieben, kann es sofort mit Ctrl + x wiederhergestellt werden. (Ctrl = Strg auf deutschen Tastaturen)

Bei *Überschreiten* der Eingabelänge wird das letzte eingegebene Zeichen rot dargestellt, es ist dann nur noch die Taste Rückschritt aktiv.

Tabulatoren werden als Punkt dargestellt. Erst im Listing nach Drücken der Return-Taste sieht man die korrekte Darstellung. z.Z. werden je Tabulator 2 Spaces ausgedruckt.

**\*)** gilt nicht für die Eingabe auf scrollenden TFT-Displays, wenn die Konsole auf com 9 umgeleitet ist.

Eingabe einer Programm-Zeile:

```
>10..inputhex"Bitte Hex-Zahl : ",A
```

nach Return-Taste verändert sich der eingegebene Text:

```
10      input hex "Bitte Hex-Zahl : ", A    der grobe Syntax-Check wurde
```

also bestanden. Andernfalls sieht es so aus: (statt Variable A testweise a eingeben)

```
>10..inputhex"Bitte Hex-Zahl : ",a  
.....x                                mit Markierung der Fehler-Stelle  
Syntax-Error
```

Input einer Hex-Zahl:

```
Bitte Hex-Zahl : 2AAFFFFF_    eine Ziffer zu viel eingegeben, mit Rücktaste löschen
```

```
Bitte Hex-Zahl : 2AAFFF_    2 Zeichen gelöscht
```

```
Bitte Hex-Zahl : 2AA31FFF    an der 5.Stelle 1 Zeichen zu viel eingefügt, Eingabe
```

hat auf Überschreib-Modus umgeschaltet, mit Ctrl + x wieder korrigieren:

```
Bitte Hex-Zahl : 2AA3FFF
```

## Funktionstasten

Die Funktionstasten 1...8 sind vom System mit folgenden Texten belegt:

F-Taste	Text	+ Return
1	dir 1	+ chr\$(13)
2	dir 2	+ chr\$(13)
3	list	+ chr\$(13)
4	save	+ chr\$(13)
5	dump0	+ chr\$(13)
6	dump8	+ chr\$(13)
7	dump9	+ chr\$(13)
8	dumpV	+ chr\$(13)

F1      F2      F3      F4      F5      F6      F7      F8  
dir 1    dir 2    list    save    dump0    dump8    dump9    dumpV

## Farben & Fonts

Die Farbpalette wurde nötig, da die verwendeten TFTs 16Bit- bzw. 24Bit- Farben darstellen können. Für den leichten Zugriff auf die Palette gibt es das Array **P()**.

### Vom System vordefinierte Farben

Farbnummer	Farbname	Farbwert 24Bit (nur ILI9488)	Farbwert 16Bit
0	schwarz	0	0
1	braun	11819520	32768
2	rot	16711680	63488
3	orange	16756736	64896
4	gelb	16776960	65504
5	gruen	65280	2016
6	blau	255	31
7	violett	13631743	47127
8	grau	8355711	33808
9	weiß	16777215	65535
10	d_braun	8527872	33024
11	d_rot	11337728	43008
12	d_orange	15233024	60288
13	d_gelb	15251712	60480
14	d_gruen	32512	992
15	d_blaue	127	15
16	d_violett	7864440	30735
17	d_grau	5197647	19049
18	silver	12632256	50712
19	magenta	16711935	63519
20	cyan	65535	2047
21	azure	32767	1023
22	mint	4109449	15793

Farbnummer ist der Index des Arrays **P()**. Man kann die Werte überschreiben, das Array **P()** wird aber nicht gespeichert.

## VT100-Farben:

0	Schwarz	8	Dunkelgrau
1	Rot	9	Hellrot
2	Grün	10	Hellgrün
3	Gelb	11	Hellgelb
4	Blau	12	Hellblau
5	Lila	13	Hellrosa
6	Türkis	14	Hellzyan
7	Grauweiss	15	Hellweiss

## Fonts

0 / 08 / 16	Tahoma 14x16 (default)	<b>Abc1</b>
1 / 09 / 17	Verdana 12x13	<b>Abc1</b>
2 / 10 / 18	Courier New 8x16 (print)	<small>Abc1</small>
3 / 11 / 19	Arial Bold 16x19	<b>Abc1</b>
4 / 12 / 20	Arial Bold 21x24	<b>Abc1</b>
5 / 13 / 21	Handel Gothic 21x22	Abc1
6 / 14 / 22	Impact 35x53	<b>Abc1</b>
7 / 15 / 23	Courier New 8x16 fett (print)	<small>Abc1</small>

## Autostart

Die Autostart-Logik:

1.) Wenn eine SD-Card im Adapter steckt, wird nach der Datei "AUTOSTART.PRG" im Root-Verzeichnis gesucht und – wenn vorhanden, gestartet.

war 1.) nicht erfolgreich:

2.) wird die Datei 0000 vom Flash-Chip geladen. Wurde sie mit Autostart gespeichert, wird sie gestartet.

oder:

3.) Ist kein Flash-Chip vorhanden, wird die im ROM befindliche Datei geladen – oder: wurde sie mit Autostart gespeichert, wird sie direkt im ROM gestartet. (Nur bei µCs mit 1MB FlashROM)

Die Autostart-Funktion ermöglicht dem Basic-Computer, wie ein Controller (ähnlich einem Arduino) zu arbeiten. Als Konsolen-Ausgabegerät sollte unbedingt Schnittstelle Nr.1 eingestellt sein! Die Reihenfolge 2.) und 3.) kann mit [config load](#) geändert werden.

## Bits und Bytes

Der ARM-M4-Prozessor ist ein 32-Bit-Prozessor. Was heisst das? Alle Speicherstellen im Prozessor sind 32 Bit breit. Alle? Nein, einige Register z.B. die Ports, einige Zähler und Konfigurationsregister sind nur 16 Bit breit:

32-Bit:

Bit-Nr. 31 -> 1001001010111000001011101011101 <- Bit-Nr. 0

16-Bit:

Bit-Nr. 15 -> 100100101011101 <- Bit-Nr. 0

Rechts ist die Bit-Position 0 und links das jeweils höchste Bit. Ein *gesetztes* Bit (=1) hat den Wert:

$2^{\text{Bitnummer}}$  ("2 hoch Bitnummer" siehe auch [Zugriff auf einzelne Bits](#))

z.B.:  $2^0=1, 2^1=2, 2^2=4, 2^3=8, 2^4=16, 2^5=32, 2^6=64, 2^7=128, 2^8=256, \dots$

Das Basic rechnet diesen Wert auch gerne aus:

```
clear  
let A.9 = 1           Bit Nr.9 setzen  
print A
```

Ein *gelöschtes* Bit (=0) hat an jeder Position den Wert 0.

32 Bits können Werte von 0... 4294967295 oder von -2147483648... 2147483647 aufnehmen.  
16 Bits können Werte von 0 .. 65535 oder von -32768... 32767 aufnehmen.

Bytes sind kleinere Einheiten von 8 Bits. Wertebereich 0... 255 oder -128...127. Mit Bytes lassen sich gut Schriftzeichen darstellen. Das Basic verwendet zur Darstellung von Schriftzeichen den ASCII-Zeichensatz mit 7 Bits (Bit-Nr.7 = 0), die Zeichen ab Nr.128 werden für Tokens verwendet. Innerhalb von *Anführungszeichen* gilt diese Einschränkung nicht – aber nicht alle Geräte verstehen die Zeichen ab Nr.128 gleich – Stichworte: Codepage 437, 850 oder [1252](#). Programmiert man mit einem Editorprogramm bitte prüfen, ob die Umlaute richtig übernommen werden. Einstellung Codierung: ANSI. (UTF-8 versteht pahlbasic nicht)

## Ports

Mit der elektrischen Aussenwelt kommuniziert der µC über die Ports. Das sind 16-Bit-Register deren einzelne FlipFlop-Ein-/Ausgänge mit den Anschluss-Pins verbunden sind. Viele der Pins können über einen internen Multiplexer auch mit anderen Prozessor-Einheiten (wie I<sup>2</sup>C, Timer oder Uart) verbunden werden. Die Portbezeichnungen sind: Port A...H. Die einzelnen Ein-/Ausgänge sind durchnummieriert von 0...15. Daraus resultiert die Pinbezeichnung z.B. Port A.0. Die Boards haben eigene Pinbezeichnungen (z.B.: J2/22). In der Beschreibung beziehe ich mich auf das 407-Mini-Board. Verwendet man andere Boards, dann bitte für die gewünschte Pin-Funktion die Prozessорpin-Bezeichnung heranziehen, die gilt immer.

Basic kennt andere Bezeichnungen für Typen als C/C++:

8Bit	16Bit	32Bit	Fliesskomma
short=int8, byte=uint8,	integer=int16, word=uint16,	longint=int32, longword=uint32	float=float32=float/double extended=float64=longdouble

64Bit Integral-Typen kann pahlbasic in den 64-Bit-Fliesskomma-Variablen nicht darstellen.

## Programmausführung im ROM

Programme werden in pahlbasic seit der ersten AVR-Version im RAM bearbeitet und ausgeführt. Dauerhaft gespeichert werden sie im Flash-Chip, auf der SD-Card oder im internen ROM. Um sie auszuführen, müssen sie vom Medium ins RAM geladen werden.

Im ROM werden Programme ausschliesslich als Speicherabbild gespeichert. Das ist ein 1:1 Abbild des RAM-Speichers. Der ARM-Prozessor kann durch ein spezielles Interface auf das ROM (fast) genauso schnell zugreifen, wie auf das RAM.

Seit Version V105A32 können Programme direkt im ROM ausgeführt werden, sie müssen nicht mehr ins RAM geladen werden. Gestartet werden Programme im ROM mit `rrun`, gelistet mit `rlist`. Das Programm wird wie bisher im RAM erstellt und gründlich getestet. Anschliessend wird es mit `save@` im ROM gespeichert. Für die Programmierung werden keine speziellen 'ROM-Anweisungen' benötigt. (Für spätere weitere Bearbeitung muss das Programm aber wieder ins RAM geladen werden z.B. mit `load@`.)

Das RAM ist jetzt frei für ein weiteres Programm vom Flash-Chip oder der SD-Card.

Eine alternative Verwendung des ROM-Speichers ist die Auslagerung von oft verwendeten Programmteilen ins ROM. Programmteile, die (vom RAM aus) im ROM aufgerufen werden, dürfen keine Zeilen-Nummern verwenden \*), da Zeilen-Nummern ja doppelt auftreten können. Es kommen *ausschliesslich* Prozeduren und Funktionen in Frage, die natürlich nur mit Label arbeiten.

Um Prozeduren im ROM (vom RAM aus) zu starten gibt es die neue Anweisung `rcall`. Die Syntax ist identisch mit der Syntax von `call`. `rcall` funktioniert auch in der umgekehrten Richtung (Programme im ROM starten Prozeduren im RAM).

Funktionen im ROM werden mit `fr Name()` (numerisch) bzw. `fs Name()` (String) aufgerufen. Die Syntax ist identisch mit ihrem jeweiligen RAM-Pendant. `fr` und `fs` funktionieren ebenfalls über Kreuz.

Auch in diesem Fall stellt man die Sammlung von Prozeduren und Funktionen zunächst im RAM zusammen als ein einziges Programm. Dieses Programm, das *nur* Prozeduren und Funktionen enthält, wird dann mit `save@` im ROM gespeichert.

User-Funktionen und -Prozeduren können jetzt (V105A32) auch von der Kommandozeile aufgerufen werden. Das Basic wird durch die eigenen Routinen erweitert.

Beide Verwendungen des ROM-Speichers verdoppeln den Programmspeicher praktisch.

Inzwischen (V1.12) gibt es eine dritte Möglichkeit: die Programmteile werden im RAM getestet, am PC zusammengestellt (Basic-Text!) und mit `rload` von SD-Card ins ROM geladen. Das ROM kann (nur) auf diese Weise jetzt Programme mit bis zu 256KByte aufnehmen. Die muss man erstmal vollschreiben. Reicht der Platz immer noch nicht? Dann kommt `lcall` ins Spiel:

Das Programm im ROM lädt während des Betriebs mittels `lcall` Programmteile nach (ins RAM) und startet sie. Nach ihrer Beendigung, fliegen sie wieder aus dem RAM und machen Platz für neue Ladevorgänge, solange die SD-Card es erlaubt...

\*) ausser am Zeilenanfang natürlich ☺.

# Hardware/Firmware

## Der STM32F407 Controller

Den STM32F407 gibt es wohl schon mindestens seit 2013. Er gehört leistungsmäßig zum Mittelfeld. Ein echtes Arbeitstier. Das Reference Manual hat 1749 Seiten im Vergleich zum ATMEGA (472 Seiten). Mit 168 MHz CPU-Takt läuft er 10-mal so schnell wie ein AVR mit 16 MHz. Effektiv ist das Basic mindestens 5-mal schneller, die Komplexität ist höher – viele Beschränkungen der AVR-Version sind überwunden.

Er hat von allem mehr als ein ATMEGA: mehr Timer, mehr I/Os, mehr Flash, mehr RAM usw. Das einzige, was er nicht hat: EEPROM. Dafür gibt es batterie-gepuffertes RAM mit unbegrenztem Speicherzugriff. Es gibt USB, Ethernet, Kamera- und Mikroprozessor-Interface, Hash- und Kryptographie-Prozessor. Davon wird nicht allzu viel vom Basic genutzt. Aber wer weiß. Die interne RTC, der Zufallsgenerator und die 12-Bit DA- und AD-Konverter auf jeden Fall. Der RAM-Speicher ist leider nicht „am Stück“. 128K für Programm und Variablen, 64K für die Arrays. Erinnert irgendwie an Commodore – wer den [CBM 720](#) noch kennt, weiß was ich meine.

Der Basic-Interpreter ist selbst auch in Basic ([MikroBasic von Mikroelektronika](#)) programmiert (wie sich das so gehört).

```
13860: sub procedure instr_while
13861: dim progcounter as ^byte volatile
13862: if pmode = 0 then perror = 1 exit end if
13863: if calculate_exp(2) = 0.0 then
13864:     progcounter = search_wend(progpointer)
13865:     if perror then exit end if
13866:     while progcounter^ > 1 inc(progcounter) wend
13867:     progpointer = progcounter
13868:     exit
13869: end if
13870: if gostack < loopdepth then
13871:     inc(gostack)
13872: else
13873:     perror = 14 exit
13874: end if
13875: returnadr[gostack - 1] = mcontadres
13876: loopvar[gostack - 1] = 40
13877: exit_adres[gostack - 1] = 0
13878: inc(loopcounter)
13879: end sub

13880: sub procedure instr_while2
13881: if calculate_exp(1) = 0.0 then
13882:     inc(loopcounter)
```

• Anweisung while  
• Test-Programmzeiger  
• im Direktmodus nicht erlaubt  
• Bedingung nicht erfüllt  
• Schleifen-Ende suchen  
• kein vend? -> Fehler!  
• ans Ende der Anweisung  
• Programmzähler setzen  
• fertig!  
• Bedingung erfüllt, Schleife starten  
• Stapel erhöhen  
• kein Platz auf dem Stapel -> fertig!  
• Rückkehr-Adresse auf den Stapel  
• while/vend Marker setzen  
• Exit-Adresse noch unbekannt  
• Schleifenzähler erhöhen  
• Anweisung while2  
• Bedingung nicht erfüllt

### < Abbildung: Anweisung while

Leider muss man das Rad ziemlich oft neu erfinden oder auch die eine oder andere HAL-Library von C++ nach Basic portieren. Auch die Unterstützung für die neueren (H7) Chips fehlt.

Den F407 Controller gibt es in mehreren Varianten mit verschiedenen RAM-, ROM-Ausstattungen und 64 bis 176 Pins. Für das Basic werden die 100- und die 144-Pin-Varianten (F407VG/F407ZG) verwendet. 144 Pins für die Version eines selbstständigen Computers mit USB-Tastatur und 800x480 Pixel TFT-Bildschirm. Neu: auch die 64-Pin-Variante F405RG.

Alle F407 Controller können *mit* und *ohne* serielllem Flash-Speicher (W25Q16 oder W25Q64) betrieben werden. Im internen ROM-Speicher kann z.Z. nur ein Programm gespeichert werden. Die Lebensdauer des internen ROMs ist mir nicht bekannt, man kann von 10.000 Schreibzyklen ausgehen. Der serielle Flash-Speicher ist mit 100.000 Schreibzyklen deutlich langlebiger. Seit kurzem (2023) gibt es 4Mbyte EEPROMS mit 500.000 Schreibzyklen (M95P32 wird vom System erkannt).

Für die Breadboard-Variante wird der STM32F411 (100 MHz, 48 Pins) eingesetzt, dem viele Features fehlen.

### Ein Wort zum Anschluss externer Komponenten

Für das ARM-Projekt gibt es ja keine fertige Platine mit allen Komponenten und Anschlüssen sowie Schutzmaßnahmen, sondern nur die chinesischen "Experimentierboards". Der Rest ist DIY, also etwas für Elektronik-Fans und Bastler.

Die ATMEGAS sind legendär robust - die ARMs sind ein wenig empfindlicher. Die Spannungen an sämtlichen Pins sollten 3,3V \*) nicht übersteigen. Der absolut maximale Strom, der auf den Versorgungsanschlüssen fließen darf, beträgt 150mA. Bei rund 100 Pins und einem Eigenstrombedarf von 70-80mA bleiben etwa 4mA für jeden der 16 Ausgänge. Beim Experimentieren: vor dem Anschluss externer Komponenten besser die Spannung abschalten – ich spreche da aus Erfahrung.

\*) viele Anschlüsse sind 5V-tolerant, aber nur, wenn sie als digitaler Eingang oder Opendrain-Ausgang programmiert sind. Mit 3,3 Volt geht man auf Nummer sicher.

## Wie kommt das Basic in den Controller-Chip?

Der Hersteller ST Microelectronics bietet umfangreiche Software zum Programmieren seiner Chips an. Downloadmöglichkeit von der Website des Herstellers. Eine Registrierung ist erforderlich.

Die einfachste Möglichkeit (ohne Programmiergeräte) ist die Übertragung mittels USB-Kabel und Verwendung der Software [STM32 Cube Programmer](#).

Dazu wird der Anschluss Boot0 (beim 407-Mini-Board am Stecker J1, Pin1) mit 3.3V verbunden, die USB-Verbindung hergestellt, Reset Taster betätigt und im Cube Programmer die Verbindung hergestellt. Dann die HEX-Datei geladen und die Programmierung gestartet.

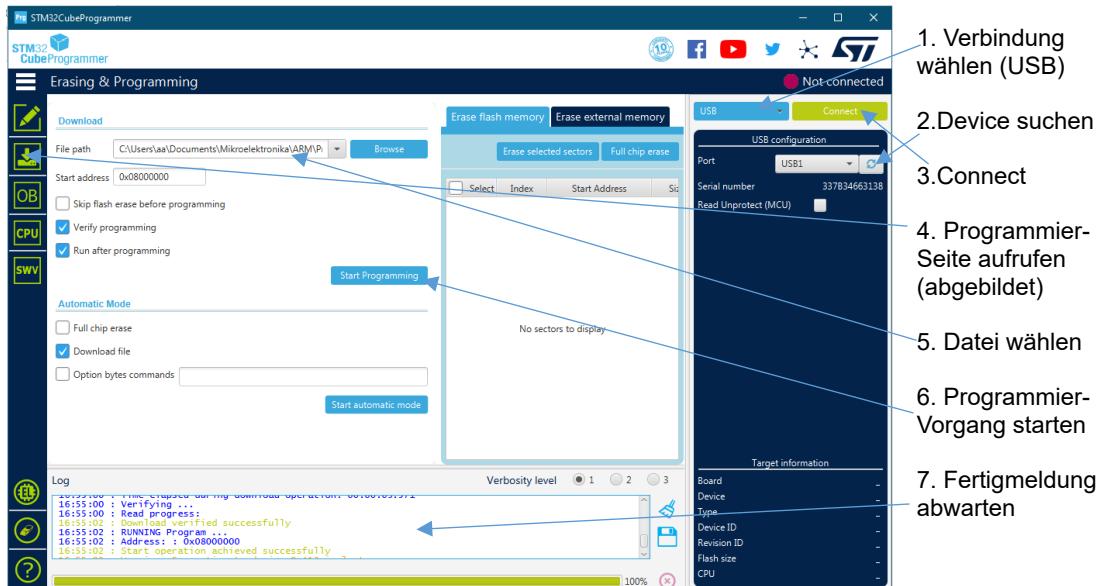


Abb. Cube Programmer

Nach der Programmierung den Boot0-Verbindungsdrat wieder entfernen und Reset-Taste drücken. Für erste Tests des Basic-Computers ist nur eine Terminal-(USB-)Verbindung nötig.

Bei den anderen Boards muss keine Drahtverbindung angebracht werden, um den Controller in den Bootmodus zu versetzen. Diese Boards haben einen zusätzlichen Taster (mit Boot oder Boot0 bezeichnet). Man betätigt diesen Taster und während man ihn festhält drückt man kurz den Reset Taster. Danach kann man den Boot-Taster loslassen. Dann wie gehabt die obigen 7 Schritte im Cube Programmer. Zum Abschluss: Reset-Taste.  
Es gibt auch Boards mit Jumpern für die Booteinstellung: Bootjumper setzen, Reset-Taste betätigen. Nach der Programmierung: Jumper in Stellung Betrieb bringen, Reset-Taste drücken.

Eine preiswerte alternative Möglichkeit der Programmierung bieten die vielen erhältlichen ST-Link V2 Clones, die mit der (nicht mehr ganz frischen) Software ST-Link arbeiten. Aber wenn man öfters programmiert, sind sie deutlich komfortabler. Ich setze J-Link-Clones ein, die auch In-Circuit-Debuggen ermöglichen und mit der IDE perfekt zusammenarbeiten.

(weitere Infos auf Anfrage)

## Die Boards

### 407-Mini-Board

Erstes unterstütztes Board ist das 407-Mini-Board. Preis vor der Chip-Krise: ca. 6 Euro.

Display: 3.5“ TFT 480x320 Pixel, Controller ILI9488, Anschluss SPI, resistives Touch-Panel  
Info Display: [http://www.lcdwiki.com/3.5inch\\_SPI\\_Module\\_ILI9488\\_SKU:MSP3520](http://www.lcdwiki.com/3.5inch_SPI_Module_ILI9488_SKU:MSP3520)

CPU: STM32F407VGT6, 1Mbyte Flash, 196KByte RAM

Info Board: <https://stm32-base.org/boards/STM32F407VGT6-STM32F4XX-M>

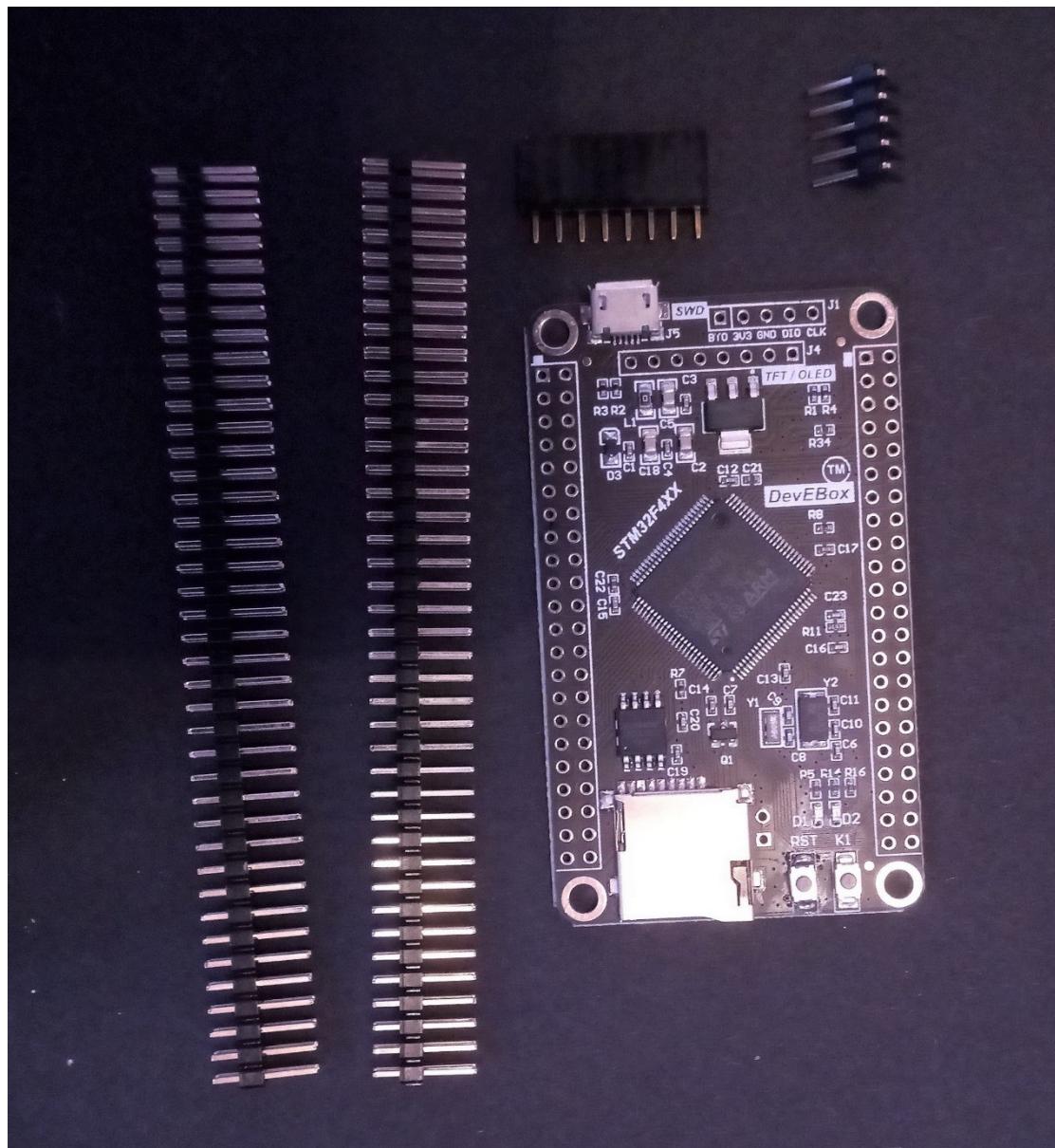


Abb. 407-Miniboard

Es können natürlich auch andere Boards mit dem STM32F407VG/ZG verwendet werden. Möglicherweise gibt es dann aber Konflikte mit Port-Anschlüssen wie z.B. dem Lebenslicht.

## 407-VE-Board

Dieses Board wird nicht mehr unterstützt. Stattdessen das 407ZG-Board. Wird demnächst vorgestellt.

Display: TFT 320x240 Pixel, Controller ILI9341, Anschluss parallel, resistives Touchpanel

CPU: STM32F407VET6, 512Kbyte Flash, 196KByte RAM

Info Board: <https://stm32-base.org/boards/STM32F407VET6-STM32-F4VE-V2.0>

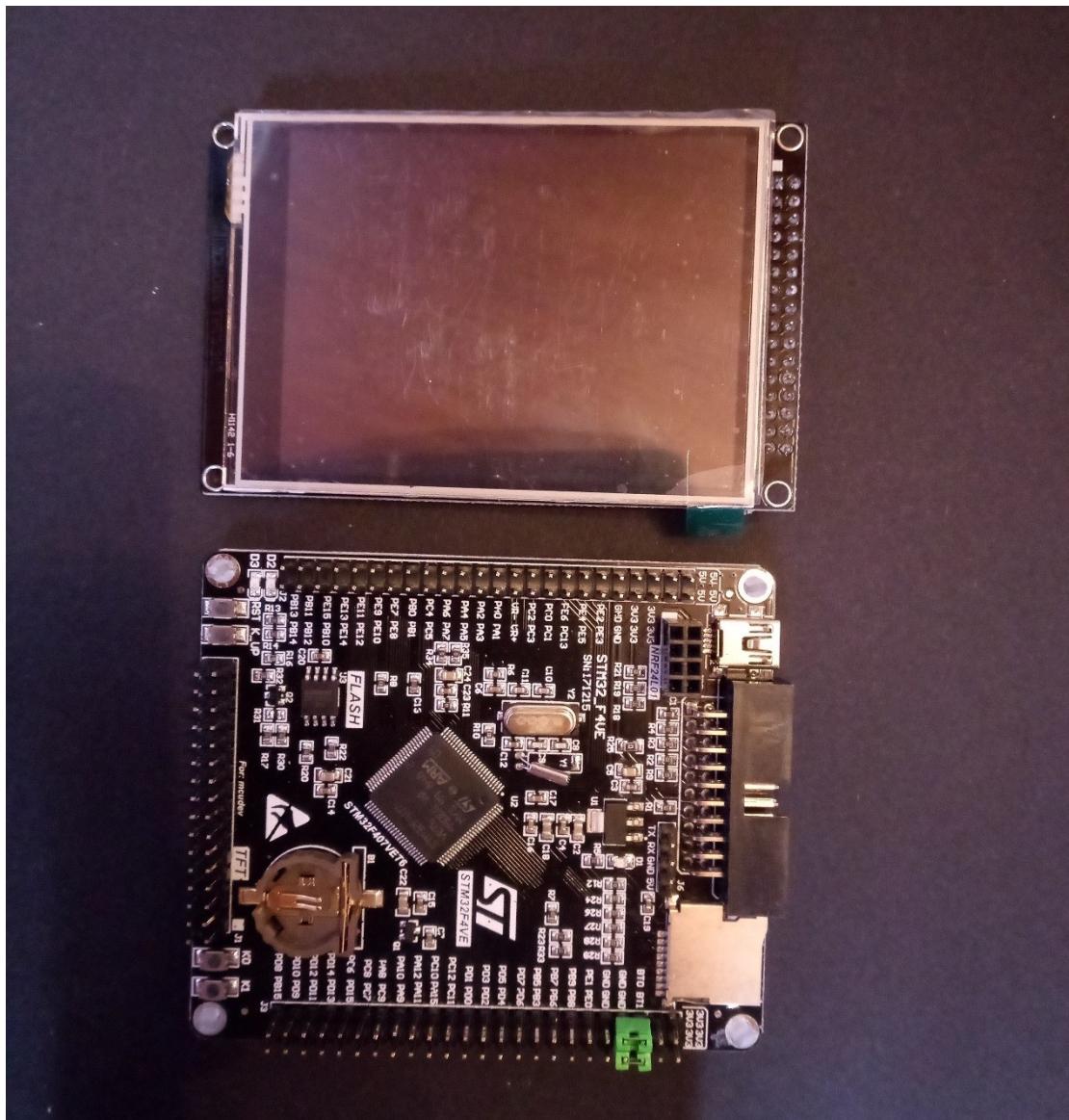


Abb. 407VE-Board

## Black Pill

Last but not least. Das Board für das Steckbrett. Preis vor der Chip-Krise: ca. 4 Euro

Display: 2.4“ TFT 320x240 Pixel, Controller ILI9341, Anschluss SPI, resistives Touchpanel  
Info Display: [http://www.lcdwiki.com/2.4inch\\_SPI\\_Module\\_ILI9341\\_SKU:MSP2402](http://www.lcdwiki.com/2.4inch_SPI_Module_ILI9341_SKU:MSP2402)

CPU: STM32F411CEU6, 512Kbyte Flash, 128KByte RAM

Info Board: <https://stm32-base.org/boards/STM32F411CEU6-WeAct-Black-Pill-V2.0>

Der Controller ist stark abgespeckt und deutlich schwächer auf der Brust. Einen ATMEGA steckt er aber locker in die Tasche. Es gibt ihn in 2 Versionen: mit seriellmem Flashspeicher und auch ohne. Vom Basic wird nur die Version ohne serielles Flash Memory unterstützt.

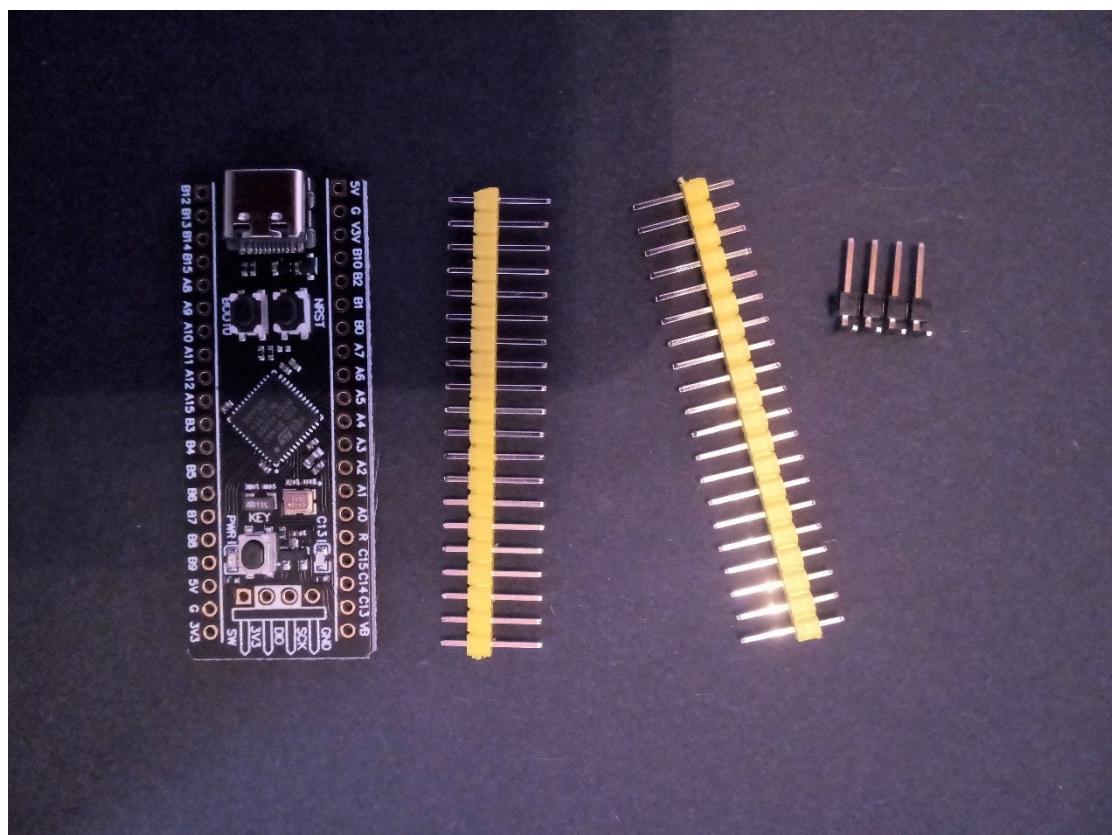


Abb. Black Pill

## Sensoren

An den I<sup>2</sup>C-Sensor-Bus können mehrere Sensoren angeschlossen werden. Beim Hochfahren scannt der Basic-Computer diesen Bus nach ihm bekannten Sensoren. Die Liste wird ständig erweitert. Es wird nur die erste Standard I<sup>2</sup>C-Adresse des jeweiligen Chips erkannt. Chips, die nicht über eine elektronische ID verfügen, werden nur nach ihrer Adresse zugeordnet. Die Adressen sind Hexadezimal.

Wie ich festgestellt habe, vertragen einige wenige Sensoren nicht einmal 100KHz (Standard) Busfrequenz. Werden diese Sensoren beim Bus-Scan erkannt, schaltet der Bus auf 20KHz herunter. Da nur wenige Bytes übertragen werden müssen, fällt diese niedrige Geschwindigkeit nicht ins Gewicht.

HTU31D Relative Humidity (rel. Luftfeuchtigkeit) und Temperatur Sensor Adress: 40

Einfacher, unproblematischer und genauer Sensor, der auch ohne Treiber, mit Basic-Bordmitteln ausgelesen werden kann. Für noch einfachere Nutzung wird er zusätzlich mit der Funktion `sensor()` unterstützt.

**BME280** Bosch-Sensor für Luftdruck, Temperatur und Luftfeuchte  
oder

**BMP280** Bosch-Sensor für Luftdruck und Temperatur Adresse: 76

Viel Mathematik erforderlich, da der Sensor keinen Prozessor eingebaut hat. Auch dieser coole Sensor wird mit der Funktion `sensor()` unterstützt, die auch die Berechnungen übernimmt. Der BME280 ist ein elektronisches Pendant zu den analogen 'Wetterstationen' von früher mit Barometer, Hygrometer und Thermometer. Den Anschluss **CSB** mit **VCC/+3.3V** verbinden – wählt I<sup>2</sup>C-Interface aus - und den Anschluss **SDO** mit **Gnd.** (wählt Adresse 0x76). Die Kombi BMP280 / HTU31D ist besonders preisgünstig.

**SDC41** Air Quality (CO<sub>2</sub>) Sensor      Adresse: 62

Der Sensor der Firma Sensirion enthält neben dem CO<sub>2</sub>-Sensor auch noch einen Temperaturfühler sowie einen Messwertaufnehmer für rel. Luftfeuchtigkeit. Das Beispiel „SCD41.PBAS“ zeigt wie man den Sensor mit Basic-Bordmitteln verwenden kann.

Ein Messwert von ca. 400ppm entspricht frischer Luft, ab 1200ppm ist die Luftqualität sub-optimal (Kopfschmerzen). Werte > 1500 bedeuten: bitte dringend lüften! Es kann auch der SDC40 eingesetzt werden.

geplant für 2025:

SDS011	Feinstaub-Sensor	UART/ser2
ADXL345	3-Axen-Motion-Sensor	Adresse: 53
INA219	Strom/Spannung-Sensor	Adresse: 80
BH1750FVI	Ambient Light Sensor (Umgebungslicht-Sensor)	Adresse: 23

## Anschluss- / Schaltpläne

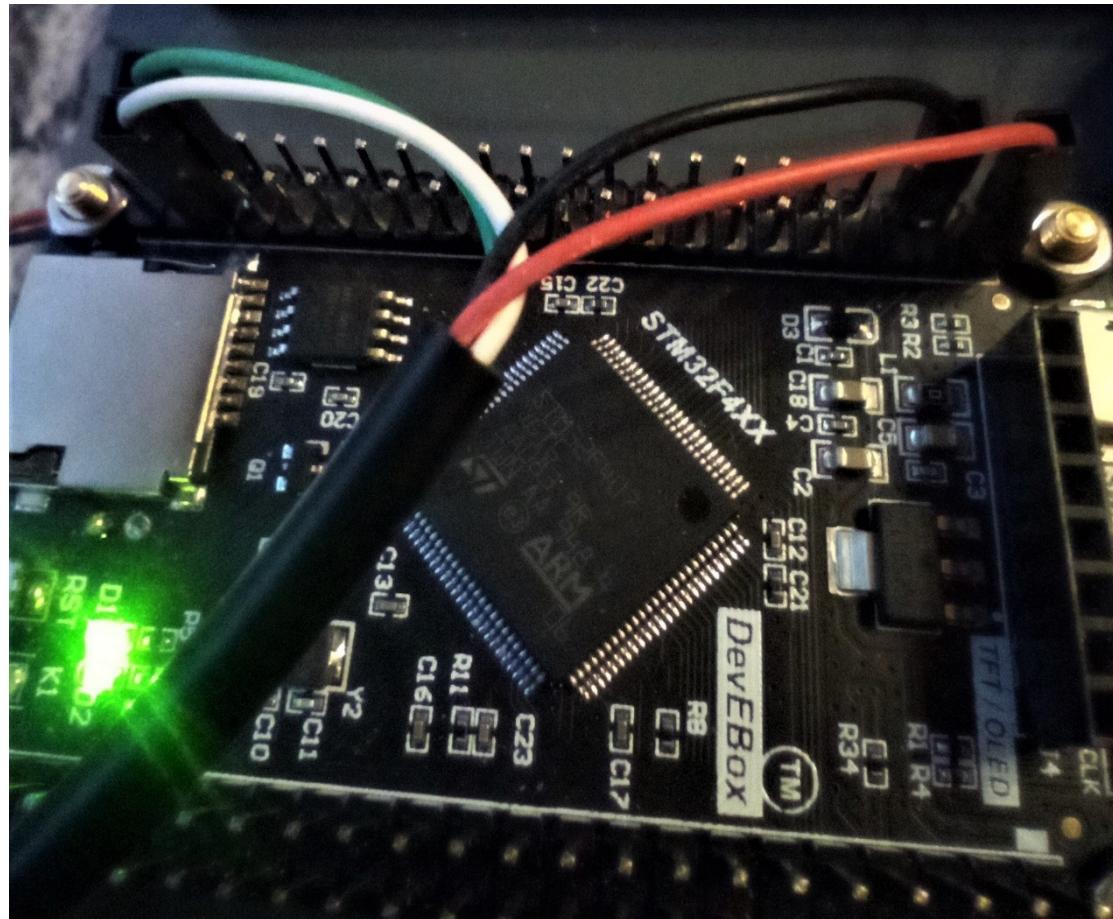
Unter den Links bei den Board-Beschreibungen findet man die Pin-Belegungen der Steckerleisten. Im Folgenden wird darauf Bezug genommen. Die vorgestellte Hardware wird vom Basic durch spezielle Anweisungen / Funktionen unterstützt.

### 1.) Das 407-Mini Board:

#### Konsolen-Anschluss:

Zunächst müssen alle Boards mit einfachen USB-Kabeln mit dem PC verbunden werden (ohne Konfiguration). Windows10/11 erkennt den Basic-Computer als serielles Gerät. In diesem Modus (USB-Konsole) können die gewünschten Änderungen konfiguriert werden. 3V Uhrenbatterie angeschlossen?

Abb. 9 serieller Konsolenanschluss:



Verwendung des *USB-Seriell*-Kabels (USB muss vorher konfiguriert werden):

auf Header 2 (J2 im Foto oben, obere Reihe ungerade Pins, Pin1 ganz rechts)

Ader	Pin	Funktion
rot:	J2-1 oder J2-2	+5V
schwarz:	J2-5 oder J2-6	Gnd
weiß:	J2-40	PB6      Uart1/TX
grün:	J2-39	PB7      Uart1/RX

Achtung! die Pegel auf den Datenleitungen (RX) dürfen **3,3 V nicht** übersteigen – bitte nachmessen. Notfalls Schutzmaßnahmen ergreifen! (Widerstände 3K3 jeweils in die beiden Datenleitungen einfügen)

## Belegung der beiden Stifteleisten

Stand: September 2023, Änderungen möglich!

<b>Header1 pins</b>	<b>(J3) rechts</b>	<b>Header2 pins</b>	<b>(J2) links</b>
1	3V3 vom int. Regler	1	+5V Versorgung
2	3V3 vom int. Regler	2	+5V Versorgung
3	Gnd Versorgung	3	3V3 vom int. Regler
4	Gnd Versorgung	4	3V3 vom int. Regler
5	Gnd Versorgung	5	Gnd Versorgung
6	Gnd Versorgung	6	Gnd Versorgung
7	VDDA - nicht verbinden!	7	PD8 Relay 9
8	VRef+ - nicht verbinden!	8	PB15 SPI2/MOSI TFT/Touch
9	PB14 SPI2/MISO TFT/Touch	9	PD10 Relay 11
10	PB13 SPI2/SCK TFT/Touch	10	PD9 Relay 10
11	PB12 Touch_CS (Chip_Select)	11	PD12 Relay 13
12	PB11 UART3 Ser4 RX / I2C2/SDA	12	PD11 Relay 12
13	PB10 UART3 Ser4 TX / I2C2/SCL	13	PD14 Relay 15
14	PE15 Touch_Interrupt_Eingang	14	PD13 Relay 14
15	PE14 TFT_CS (Chip_Select)	15	PC6 UART6/TX GPS/Ser.3
16	PE13 TFT_RS (Daten/Kommando)	16	PD15 Relay 16
17	PE12 TFT_BLED (Beleuchtung)	17	PC8
18	PE11 diginp 12	18	PC7 UART6/RX GPS/Ser.3
19	PE10 diginp 11	19	PA8
20	PE9 diginp 10	20	PC9
21	PE8 diginp 09	21	PA10 WS2812B LED-Anschluss
22	PE7 diginp 08	22	PA9
23	PB1 TFT_RST (Reset)	23	PA12 DP USB für Keyboard/Konsole
24	PB0 SD-Card_CS	24	PA11 DM USB für Keyboard/Konsole
25	PC5 RS485-RTS	25	PC10 UART4/TX MIDI/Ser.2
26	PC4 record Signal-Input (Mono)	26	PA15 Flash CS - nicht verbinden!
27	PA7 DS18B20-2 Temp.-Sensor	27	PC12 USB-Powerswitch für Keyboard
28	PA6 DS18B20-1 Temp.-Sensor	28	PC11 UART4/RX MIDI/Ser.2
29	PA5 DAC2 Audio links	29	PD1 Relay 2
30	PA4 DAC1 Audio rechts	30	PD0 Relay 1 (relay.0)
31	PA3 PWM3	31	PD3 Relay 4
32	PA2 PWM2	32	PD2 Relay 3
33	PA1 Lebenslicht LED auf Board	33	PD5 Relay 6
34	PA0 PWM1	34	PD4 Relay 5
35	PC3 adc4	35	PD7 Relay 8
36	PC2 adc3	36	PD6 Relay 7
37	PC1 adc2	37	PB5 SPI1/MOSI Flash-ROM
38	PC0 adc1	38	PB3 SPI1/SCK Flash-ROM
39	PC13 ext. Interrupt zukünftige Erw.	39	PB7 UART1 Ser1 RX Konsole
40	PE6 diginp 07	40	PB6 UART1 Ser1 TX Konsole
41	PE5 diginp 06	41	PB9 I2C1/SDA Sensorbus Daten
42	PE4 diginp 05	42	PB8 I2C1/SCL Sensorbus Takt
43	PE3 diginp 04	43	PE1 diginp 02
44	PE2 diginp 03	44	PE0 diginp 01 (diginp.0)

### rechts

(Blick auf das Board, Bestückungsseite, SD-Card-Slot links unten)

### links

## Anschluss des Displays

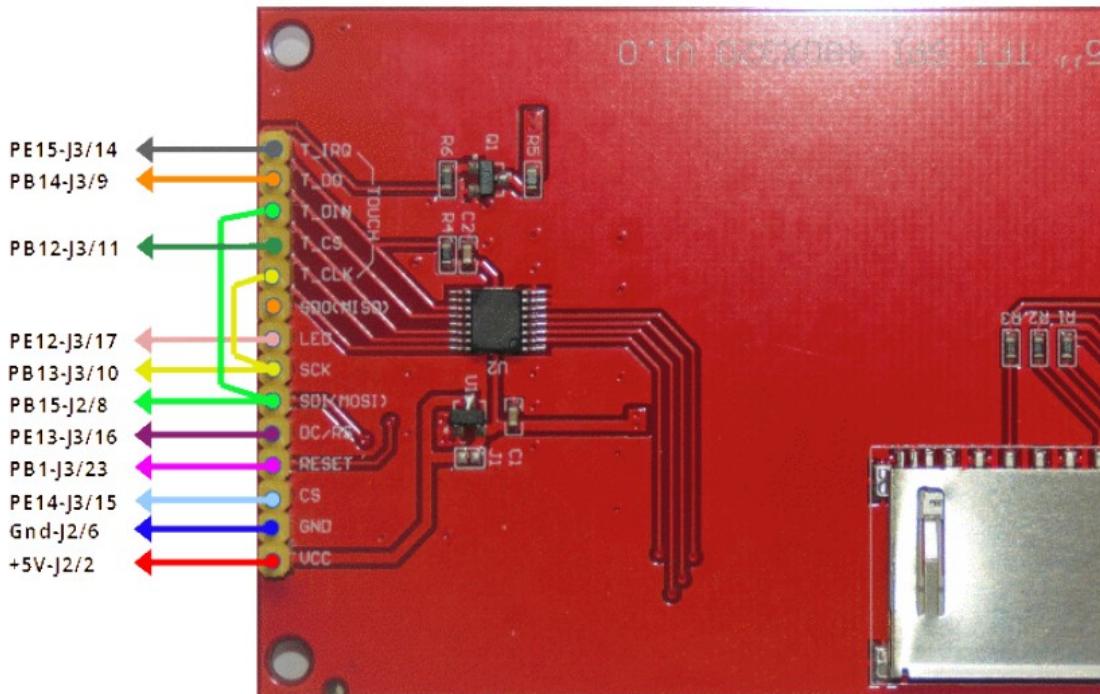


Abb. 10 Rückseite Display-Platine: ILI9488, beim ILI9341 auch T\_DO mit SDO verbinden

Auf dem Display-Board ist ein Pegelwandler sowie ein 3.3V-Regler. Einstellung siehe [config](#). Es gibt auch Displays ohne Regler. Dann VCC mit 3.3V verbinden!!!

## 2.) Das 407ZG-Board

Das 407ZG-Board ist mit dem Prozessor STM32F407ZGT bestückt. Der grösste Unterschied: der Prozessor hat mehr Pins. Die zusätzlichen Pins werden wie folgt benutzt:

Port F ist der parallele Datenport (16Bits) für das TFT-Display mit SSD1963 Graphik-Controller. Das Display wird dadurch deutlich schneller. Der Port kann evtl. auch für andere Aufgaben 'missbraucht' werden.

Port G ist jetzt der 'Relay'-Port, der Port mit den Steuerausgängen. Es bleibt bei 16 Ausgängen.

Bei einigen Pins wird die Funktion geändert:

Port E ist jetzt komplett digitaler Eingangs-Port. Es gibt also 16 digitale Eingänge.

Keine Änderung bei seriellen Schnittstellen, I2C- und SPI-Interface, DAC-Ausgängen.

Es gibt 4 PWM-Ausgänge. Die Anzahl der Analogeingänge wird mindestens 4 betragen.

Die erste Sd-Card wird über den SD-Card-Anschluss (SDIO anstatt SPI) betrieben, was die Kompatibilität mit einigen Fabrikaten verbessert sowie die Datenübertragung beschleunigt. Die zweite SD-Card - wie gehabt - über den SPI. Kopieren zwischen den Drives wird z.Z. nicht unterstützt.

Die Firmware ist noch in Arbeit, wird aber nicht veröffentlicht. Es wird µC-Boards geben mit aufgespieltem Basic. Die Firmware für das 407miniBoard (ab V105A36) läuft aber auch auf dem F407ZG. Die Anweisungen [set relay](#) und [reset relay](#) können die zusätzlichen Portpins ansprechen.

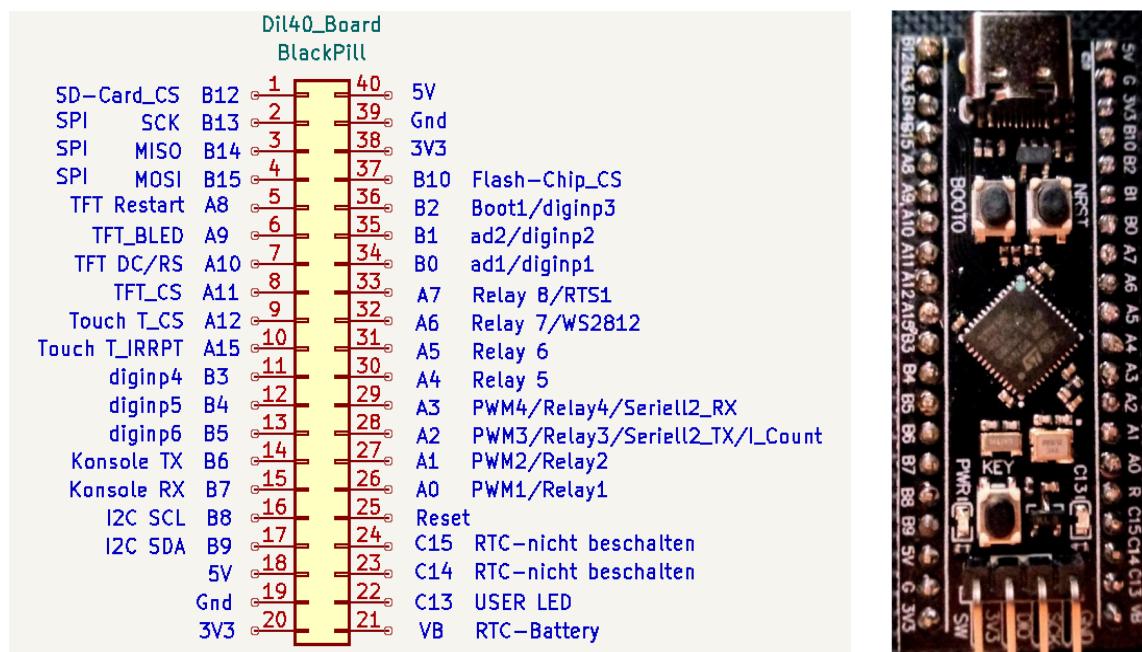
### 3.) BlackPill

BlackPill ist ein kleines Board im DIL40 Format – dadurch Steckboard (Breadboard) tauglich. Der Prozessor ist ein STM32F411CE mit 512KB Flash ROM und 128KB RAM, läuft mit 96MHz. Auch eine Version für den STM32F401CD (mit 378KB Flash und 96KB RAM sowie 84MHz) ist in Arbeit (auf diesem Chip können Programme nicht im Flash-ROM gespeichert werden). Es wird die Verwendung der Originalboards von WeAct Studio (mit 25MHz Quarz) empfohlen. Die Beschreibung bezieht sich auf die Board-Version 3.0. Die Boards mit aufgelötetem serielllem Flashspeicher werden nicht unterstützt, es kann jedoch extern via SPI serieller Flash-/EEPROM-Speicher angeschlossen werden (die Pinbelegung von WeAct passt nicht zu meiner Konfiguration). Die Boards erhalten jeweils eine eigene Firmware.

Es fehlt Batterie-gepuffertes RAM (ausser den 80Bytes RTC-RAM für die Konfiguration mit config) - es gibt also auch kein Array Z(). Es fehlt der Zufallsgenerator (rand) ist daher die übliche Softwarelösung). Es gibt eine (2) serielle Schnittstelle(n), einen I2C- sowie einen SPI-Bus. Der USB-Anschluss wird nicht unterstützt. Das Textarray T() fehlt. Der µC hat keinen Digital-/Analog-Wandler, also auch keine Soundwiedergabe (und kein Array S()). OneWire-Temperatur-Sensoren werden nicht unterstützt.

TFT-Display und SD-Card sind aber in Funktion und eine RTC sowie einen ADC hat der Chip auch. Die weit geringere Anzahl Pins wurde nach meinem Geschmack auf die verschiedenen Hardwarefunktionen (teilweise Mehrfachbelegungen) verteilt, z.B.: 4 (6) digitale Inputs, 2 analoge Inputs, 4 (8) digitale Outputs, 4 PWM-Ausgänge.

Das Basic ist identisch mit den 'grossen' Lösungen.



Pin-Belegung für das BlackPill-Board, Breadboard-Beispiele folgen demnächst

Die Systemvariable relay entspricht beim F401/F411 dem Port A, die Systemvariable diginp dem Port B, sonst gilt das beim F407 Gesagte.

Für die TFTs sollte wegen der geringeren SPI-Geschwindigkeit auf die kleineren (320x240) Displays mit ILI9341 zugegriffen werden. Die Pins der PWM-Ausgänge lassen sich auch anderweitig z.B. als Relais-Ausgänge, als Impuls-Messeingang oder 2. serielle Schnittstelle nutzen. Die Pins der Analog-Inputs lassen sich auch als digitale Eingänge betreiben.

## Schnittstelle 2 aktivieren:

Es reicht, die serielle Schnittstelle `com2` zu konfigurieren z.B.: `config com=2, 9600` und nach dem nächsten Reset werden die *PWM-Ausgänge* `PWM3` und `4` abgeschaltet und stattdessen das serielle Interface `com2` auf die Port-Pins A2 und A3 gelegt. Man kann nur die Übertragungsgeschwindigkeit einstellen. Es kann auch für MIDI oder GPS genutzt werden. Die entsprechenden Anweisungen/Funktionen sind nach dem Reset aktiv (natürlich geht GPS und MIDI nicht gleichzeitig über ein Interface ☺).

`config com=2, 0` ...und das Interface ist nach dem Reset wieder weg.

## Messeingang:

Auf dem BlackPill-Board gibt es den Eingang `diginp6` nicht, die Anweisungen `set diginp 6 as fcount` und `set diginp 6 as icount1` sowie `set diginp 6 as icount2` funktionieren trotzdem und schalten diese Eingangsfunktionen auf den *PWM-Ausgang* `PWM3` (Port-Pin A2). `reset diginp 6` stellt die vorherige Pin-Funktion wieder her. Die Anweisungen `set` und `reset` wirken sofort ohne Hardware-Reset.

Wird die äussere Beschaltung gestört, wenn der Messeingang vorher ein (PWM-) Ausgang war (z.B. Ausgang trifft auf Ausgang), dann hilft:

`config pwm3 as diginp` ...und Port-Pin A2 ist nach dem Reset sofort ein Eingang,

`PWM3` entfällt.

## Relais-Ausgänge 1...4:

Werden nicht alle PWMs benötigt können sie in Relais-Ausgänge umgewandelt werden:

`config pwm1 as relay` ...wandelt `pwm1` in `relay 1` um,

`pwm2...4` können analog in `relay 2` ...`relay 4` konfiguriert werden.

`config pwm1 as pwm1` ...macht die Umwandlung rückgängig.

## Digitale Eingänge 1...3:

Die Analogeingänge `ad1` & `ad2` werden wie folgt umkonfiguriert:

`config ad1 as diginp` ...wandelt `ad1` in `diginp 1` um,  
`config ad2 as diginp` ...wandelt `ad2` in `diginp 2` um,

`config ad1 as ad1` ...macht die Umwandlung rückgängig.  
`config ad2 as ad2` ...macht die Umwandlung rückgängig.

Boot1 muss nicht konfiguriert werden. Es ist bereits digitaler Eingang (`diginp3`). Er hat nur eine Besonderheit: die Boot1-Funktion. Für diese Funktion ist auf der Platine ein Widerstand von 10K nach Masse eingelötet, weshalb es keinen Sinn macht, einen Pullup für diesen Eingang zu programmieren.

## Flash-ROM:

Das BlackPill-Board gibt es in 2 Varianten: mit F411CE-Prozessor sowie mit dem F401CD. Der F411CE hat ausreichend ROM, um dort (wie beim F407) Programme oder Funktionen zu speichern. Der F401CD hat das nicht. Die entsprechenden Anweisungen funktionieren nicht. Es müssen Boards mit 25MHz-Quarz verwendet werden (8MHz-Version auf Anfrage). Das BluePill-Board sowie Boards mit STM32F401CC sind für pahlbasic nicht geeignet.

## Erste Schritte mit dem Basic-Controller

Hier entsteht demnächst eine kleine Anleitung für den Start mit einem µC.

Damit das Board auf dem Breadboard läuft, müssen die Stifte von Pin 23 und 24 (C14 und C15) entfernt (abgeknipst) werden. Der RTC-Oszillator kommt mit den Kapazitäten der Breadboardkontakte nicht klar. Die Pins werden aber eh nicht gebraucht.



#### 4.) Das DIY-More-Board

Wegen mehrerer Anfragen, ob [dieses Board](#) geeignet sei für pahlbasic, habe ich es einer Prüfung unterzogen. Ergebnis:

Es hat den gleichen Prozessor wie das 407Mini-Board und auch dieselbe Quarzfrequenz (8 MHz) und pahlbasic läuft natürlich darauf. Es fehlt aber das serielle Flash-ROM (kann man extern nachrüsten: W25Q16) und wie es aussieht: leider auch eine Anschlussmöglichkeit für eine Uhrenbatterie. Daher muss das gesamte Modul ständig bestromt sein – z.B. mit einem 3.7V Lithium-Handy-Akku. 'Ausschalten' kann man dieses Modul dann nur mit der eigens für diesen Zweck hinzugefügten Anweisung [sleep](#), die aber nur den Prozessor in den Schlaf versetzt – für das Abschalten evtl. vorhandener Peripherie (TFT-LCD, Relais usw.) ist der User verantwortlich. Die User-LED ist an PEO angeschlossen – diginp1. Mit diesen Einschränkungen ist das Board für Bastler brauchbar und wer es 'rumfliegen' hat, mag es gerne nutzen.

Zum Programmieren der Firmware, muss der Bootjumper gesetzt sein:



< Jumper in Boot Position

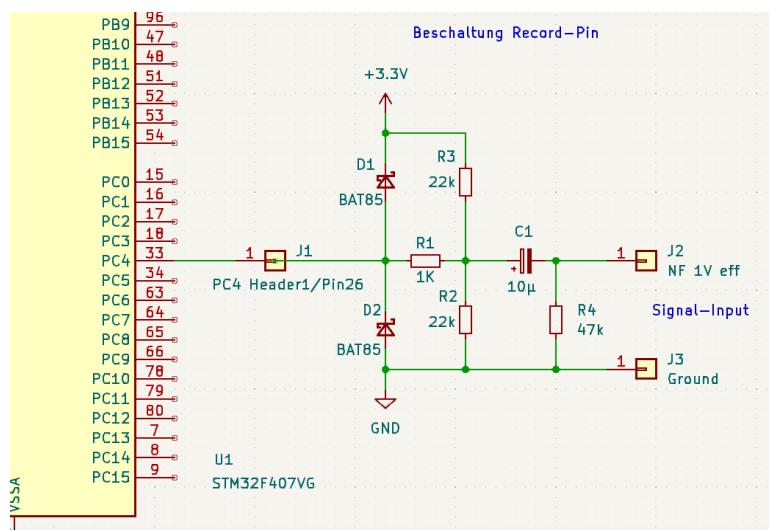
den Akku mit Diode (1N4004) an 3.3V anschliessen. (nicht im Bild)

Die Steckleiste wurde auf der Unterseite montiert (Beschriftung)



< Jumper in Betriebs-Position

## record-Pin-Anschluss



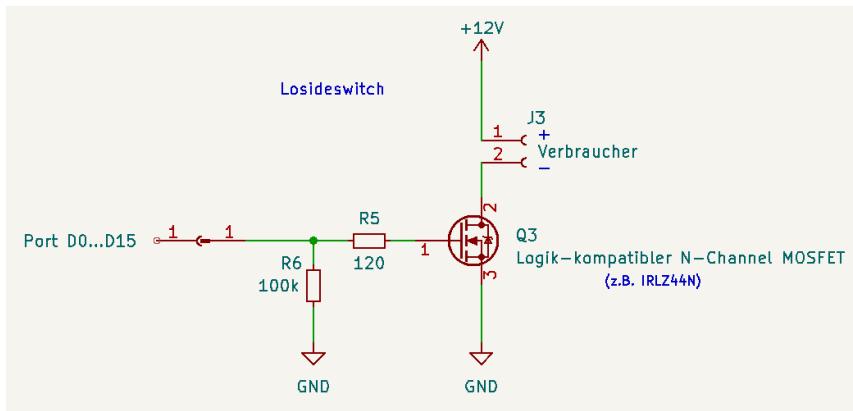
Anschluss-Bild Aufnahme-Pin (nur F405/7 μCs)

Minimale Beschaltung des NF-Aufnahmeverbindungsstücks, rein passiv. Eingangswiderstand ca. 10k. Ein Mikrofon lässt sich z.B. mit Hilfe des Mikrofonverstärkers (siehe Website/Projekte) anschliessen.

Die Schottky-Dioden auf keinen Fall weglassen! Man kann auch Schottky-Gleichrichter-Dioden wie die 1N5817 verwenden.

## Port-Ausgänge

Die Portausgänge sollten nicht mit mehr als 4 mA belastet werden (wegen des max. Stroms auf den Versorgungsanschlüssen der CPU). Noch sicherer wäre ein max. Strom von 1 mA. Damit kann man natürlich allenfalls stromsparende LEDs einschalten. Für etwas mehr Power kann man die 16 "Relais"-Ausgänge mit zusätzlichen Transistoren etwas 'verstärken'. Die folgenden Beispiele sind galvanisch *nicht* vom CPU-Board getrennt. Beim Anschluss ist also gewisse Vorsicht geboten. Der einfachste Powerschalter ist ein Transistor, der nach Masse schaltet:



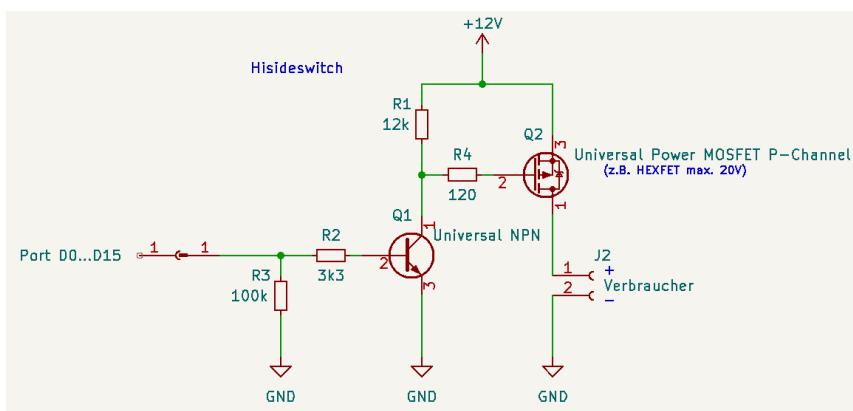
Da die Transistorwerte etwas streuen, jeden Transistor vor Einbau testen, ob er bei 3V Gatespannung sicher schaltet (garantiert sind nämlich nur 4V). Diese Schaltung funktioniert auch mit höheren Spannungen (bis 40V und < 5A). Und: 16 Stück nehmen nicht viel Platz ein.

Abbildung: LoSideSwitch Port D0 entspricht Relay1, D15 entspricht Relay16

Dieser Schalter ist optimal, wenn unterschiedliche Spannungen geschaltet werden müssen. Jeder Transistor kann "seine eigene" Spannung schalten. Da nur ein Transistor involviert ist, ist die Schaltgeschwindigkeit hoch. Der Verbraucher darf selbstverständlich auch ein Relais oder Schütz sein für noch höhere Ströme oder um Wechselspannungen zu schalten. R5 ist als Schutz vor Stromspitzen bei Potentialwechsel und hohen Gatekapazitäten vorgesehen – eventuell auf 1k erhöhen.

Profis können die Ausgänge des Controllers natürlich auch in Opendrains umkonfigurieren und den Transistor mit 5V sicher ansteuern.

Ist für sämtliche Verbraucher eine einheitliche Spannung vorgesehen (gilt nur für integrierte Mehrfachschalter), kann der Schalter auch im positiven Ast werkeln:



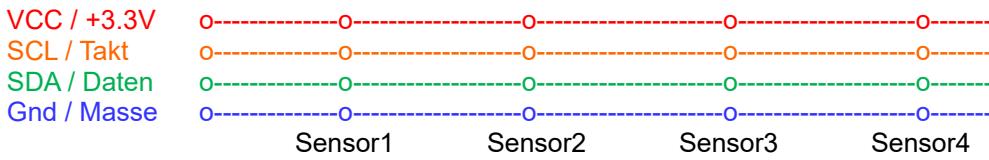
Die P-Kanal MOSFETs eignen sich oft nur bis 20V. Sämtliche PIN-Angaben sind nicht bindend. Bitte dem Datenblatt des jeweiligen Transistors entnehmen. Die 2 Transistoren schalten etwas langsamer als die obere Schaltung. Der bipolare Transistor schaltet ab 0.7V.

Abbildung: HiSideSwitch Universal NPN z.B.: BC807

Für beide Schaltungsvarianten gibt es integrierte Lösungen (meistens für mittlere Ströme 500mA...1A). Auch galvanisch getrennte Schalter (Solid-State-Relays für Gleich- oder Wechselspannung) sind einsetzbar. Bitte immer die max. Strombelastbarkeit des Controllers im Auge behalten.

## Der Sensor-Bus

Der I2C-Sensor-Bus besteht aus 4 Leitungen:



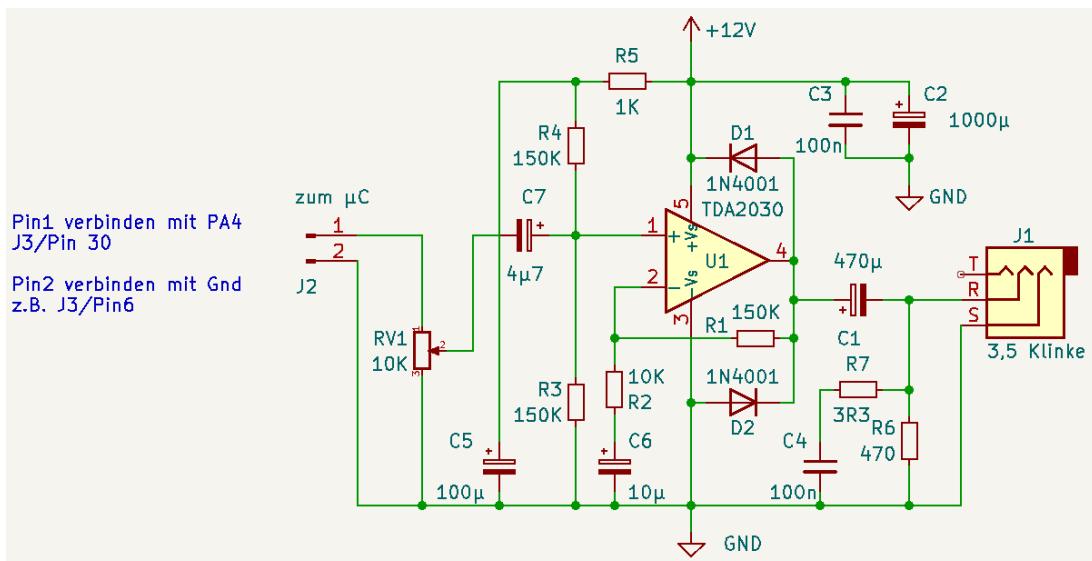
Jeder Sensor hat Anschlüsse mit den Bezeichnungen **VCC**, **Gnd**, **SDA** und **SCL**, die mit den gleichnamigen Anschlässen des Sensor-Busses verbunden werden. Am rechten Ende des Busses werden die Leitungen **SDA** und **SCL** bei Bedarf \*) jeweils mit 10KOhm-Widerständen nach **VCC / +3.3V** verbunden. Die linke Bus-Seite wird mit den entsprechenden Anschlässen der Stiftleiste Header2 (J2) des CPU-Boards verbunden:

VCC → J2/3, SCL → J2/42, SDA → J2/41, Gnd → J2/5

**Stiftreihe J2 ist links (mit Blick auf die Komponentenseite des Boards, SD-Karten-Slot links unten).**

**\*)** Auf vielen Sensor-Breakout-Boards sind die Widerstände bereits integriert.

## Audio-Anschluss:

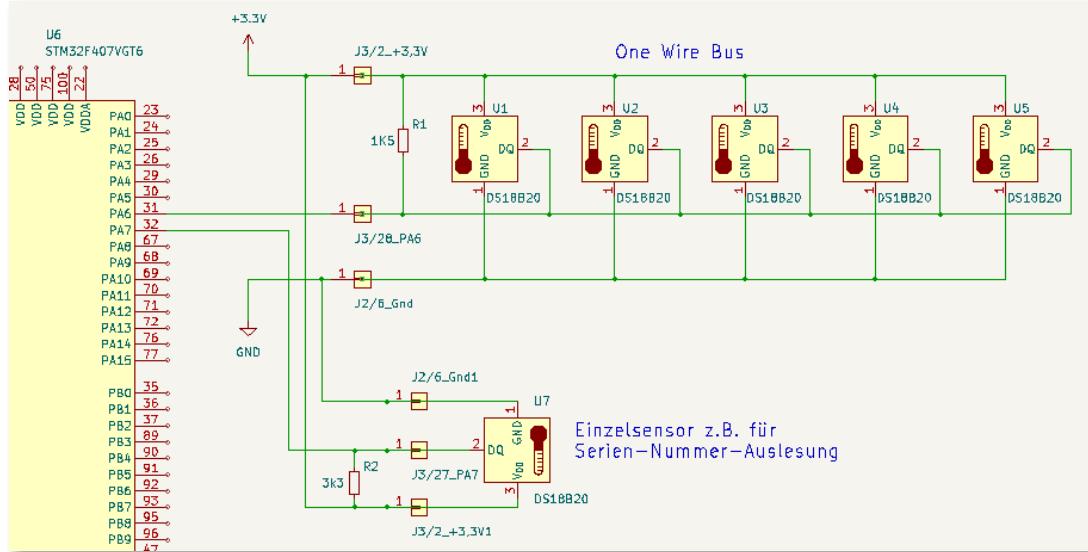


*Es wird der Anschluss eines Verstärkers empfohlen: Darstellung Beispielschaltung rechter Kanal*

Ich habe auf meinem "Entwicklungsboard" 2 kleine Verstärkermodule für die Audioausgabe eingesetzt. Die Schaltung dürfte in Etwa der obigen entsprechen. Die winzigen Kühlkörper lassen nur Kopfhörer zu. Die Audio-Ausgänge am µC sind Port-Pins PA4 und PA5 (Stereo). (beim 407-Mini-Board J3 / Pin 30 und Pin 29). Es funktionieren aber auch andere kleine Breakoutboards oder Eigenbauten mit kleiner Leistung mit min. 10K Eingangsimpedanz. Die Ausgangsspannung des µC-Boards liegt bei ca. 1V eff. und hat eine Gleichspannungskomponente von 1,6V, die abgetrennt werden sollte. (macht C7)

## DS18x20 One-Wire-Temperatur-Sensoren-Bus

Schaltplan des One-Wire-Busses:

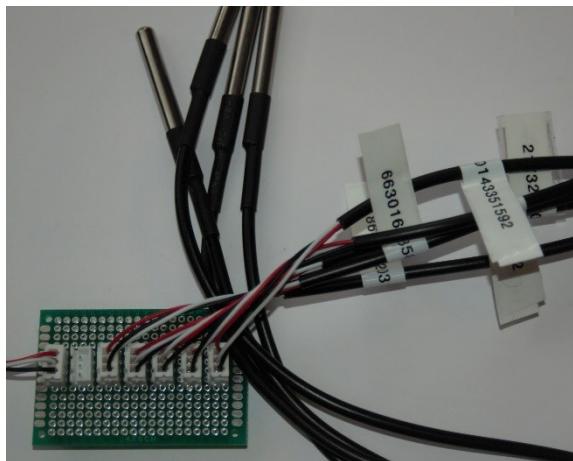
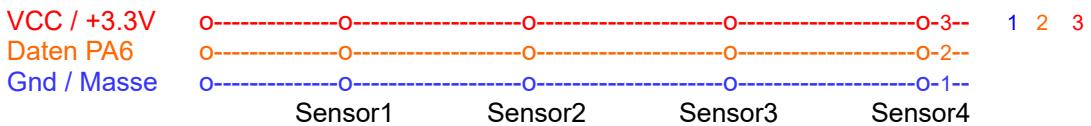


Bitte beachten: U7 ist rotiert dargestellt (im Vergleich zu U1...U5), andere Anschlußfolge beachten!

DS18S20



Der One-Wire-Bus besteht (wie der Schaltplan zeigt) aus 3 Leitungen:



Daten (J3/28) und VCC mit 2k7 \*) Widerstand verbinden. Bei den Sensoren mit Anschlusskabel sind die rote Leitung mit VCC, die weiße Leitung mit Daten und die schwarze Leitung mit Gnd zu verbinden.

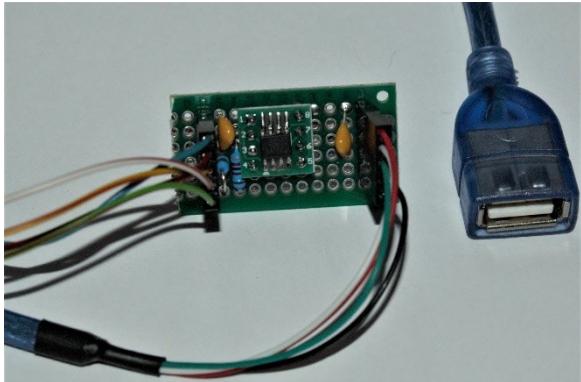
An PA7 (J3/27) wird ein einzelner Sensor entsprechend angeschlossen.

\*) Abhängig von der Anzahl der Sensoren. Funktioniert die Übertragung nach Hinzufügen weiterer Sensoren nicht mehr, Widerstandswert verringern (1k funktioniert auch noch)

Praktische Ausführung eines Testaufbaus

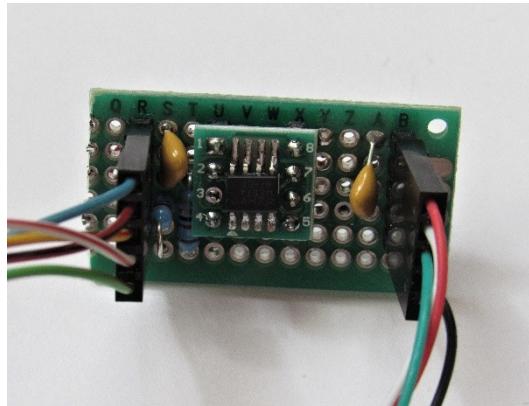
## USB-Keyboard

Ein Computer ohne Tastatur? Geht natürlich nicht. Es sollte natürlich eine handelsübliche USB-Tastatur verwendbar sein. Alles was es dazu braucht, ist ein kleines Zusatzplatinchen:



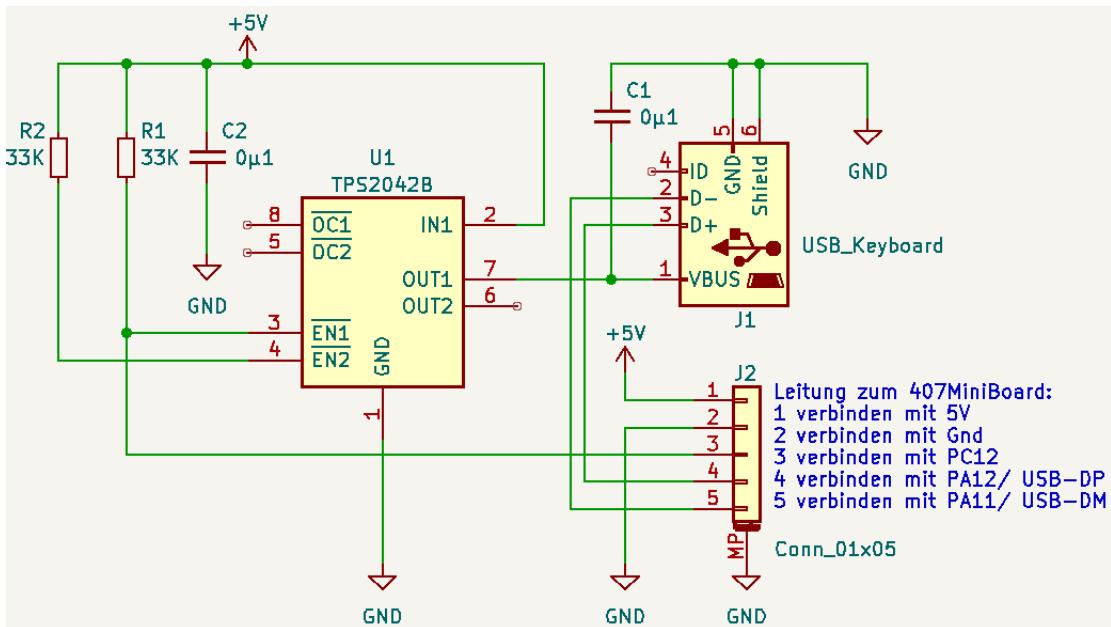
Dieses kleine Platinchen ist notwendig, damit der USB-Host-Anschluss die 5V Spannungsversorgung ein- und ausschalten kann. Es wird zwischen den USB-Anschluss (auf dem Controllerboard) und das Keyboard geschaltet. (Damit sorgt die Firmware für einen definierten Start des Keyboards. Bei einigen Fabrikaten muss mehrmals gestartet werden.)

Leider ist der TPS2042B nur als SMD erhältlich. Ich habe ihn auf eine kleine SOP8-Adapterplatine gelötet, die dann wie ein Through-Hole-Bauteil einsetzbar ist:



Der USB-Anschluss muss für den Einsatz des Keyboards konfiguriert werden oder auch nur eingeschaltet.

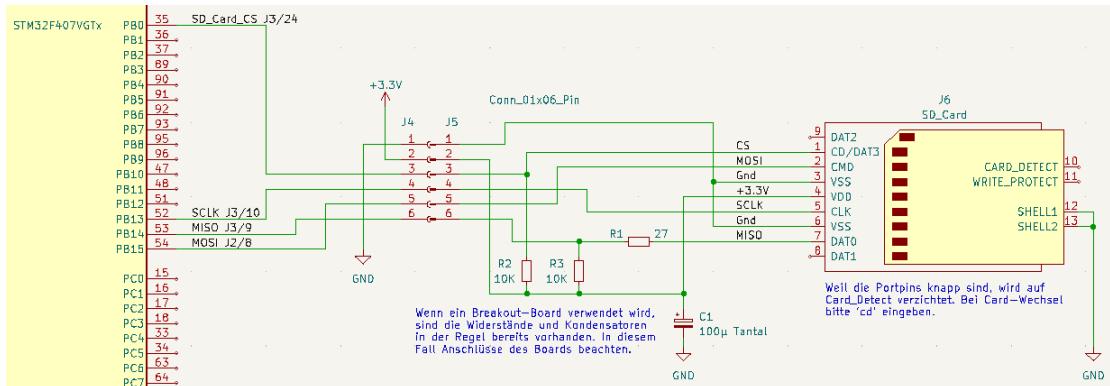
Der TPS2042B ist lediglich ein (2) Schalter, der die 5V-Spannung für den USB einschaltet. Er enthält praktischerweise auch noch einen 500mA Strombegrenzer und einen Übertemperaturschutz. Im Prinzip täte es auch einfach nur ein Transistor. An OC1 kann eine LED mit Vorwiderstand angeschlossen werden (gegen +5V), um Kurzschlüsse oder Überströme anzudeuten:



Die Beschaltung des kleinen Platinchens – es kann auch ein TPS2041B eingesetzt werden.

Natürlich kann der USB nur einem Herrn dienen. Die Konsole wird auf Schnittstelle 1 geleitet. Die Funktion wird nur für das K340 und das K400+ von logitech (ohne die Mausfunktion) garantiert. Neuere Keyboards sind problematisch, die warten auf mir unbekannte Daten.

## SD-Card

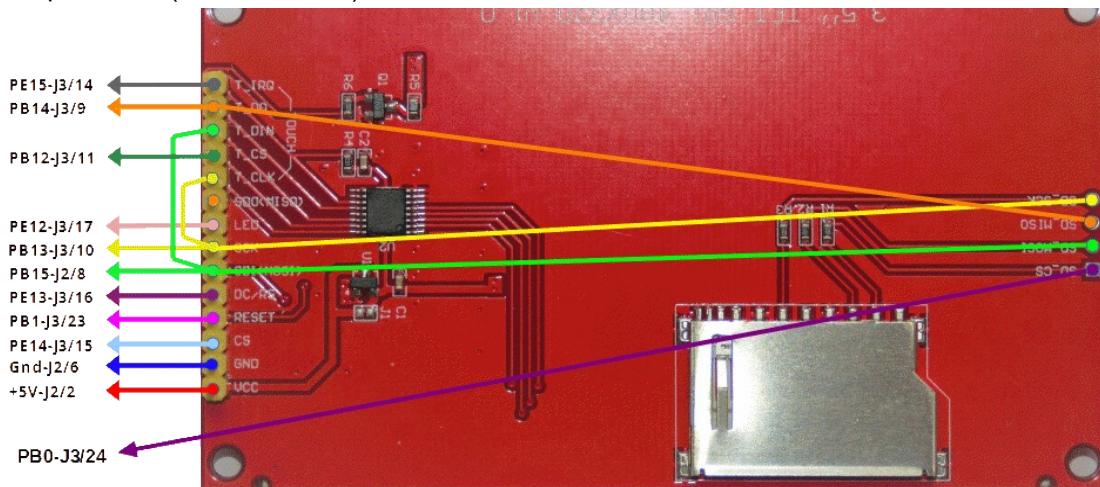


Anschluss der SD-Card (Signal-Bezeichnungen beziehen sich auf das 407Mini-Board)

Die SD-Card benötigt im Prinzip nur den SPI-Bus (Nr.2) sowie ein Chip\_Select-Signal und Spannungsversorgung. Die Busleitungen so kurz wie möglich halten. Auf das Card-Detect-Signal wird verzichtet, d.h. ein Kartenwechsel wird nicht automatisch erkannt. Viele Breakout-Boards haben dieses Signal auch nicht.

Ist die SD-Card beim Hochfahren des Basic-Computers eingesteckt, wird sie detektiert und initialisiert. Nach einem Wechsel der Karte bitte cd auf dem Terminal eingeben.

Es kann auch der SD-Card-Adapter auf dem TFT-Display verwendet werden, der SD-Slot auf dem µC-Board (407Mini-Board) darf leider nicht benutzt werden:



Rückseite Display - Nachverdrahtung des SD-Card-Adapters (schräge Linien)

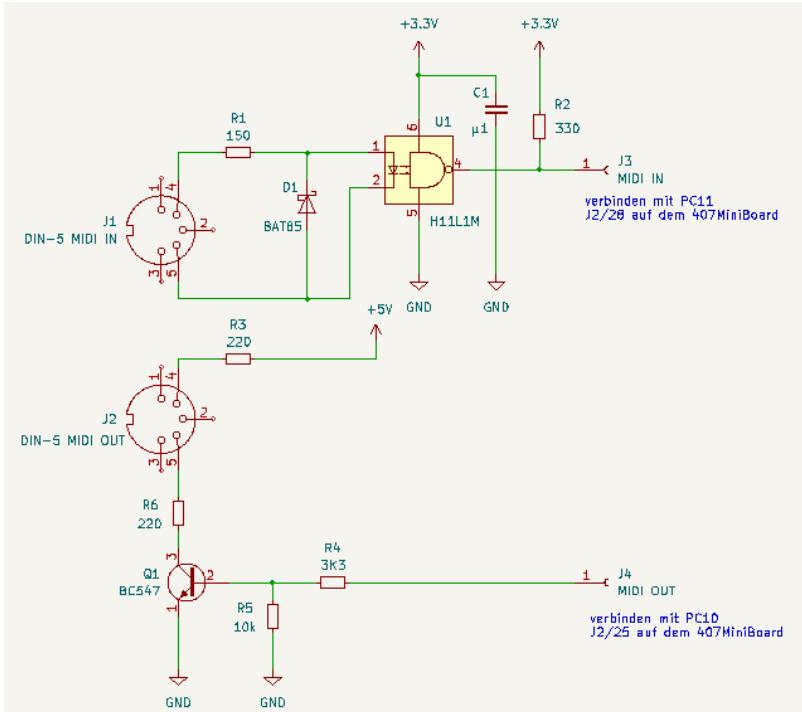
Zwischen +5V und Gnd einen Kondensator von 47µF einfügen (direkt am Display-Stecker).

Es werden SD-Cards und SDHC-Cards von 2...32 GByte, die mit FAT32 formatiert sind, unterstützt, z.B.: SanDisk Ultra 32GByte. Man kann sie mit dem PC formatieren (Schnellformatierung, Grösse der Zuordnungseinheiten=Standard) oder das den Basic-Computer machen lassen – dauert aber deutlich (Minuten!) länger. (siehe format)

Programme können (nur) auf der SD-Card auch als Basic-Text (.PBAS) gespeichert und geladen werden, (.PRG=Programm-Speicher-Abbild geht natürlich auch).

## MIDI

Die Standard MIDI-Schnittstelle zum Anschluss externer Musikinstrumente



*Hardware der Standard MIDI-Schnittstelle - statt Transistor kann auch ein TTL-Inverter eingesetzt werden*

unbedingt den angegebenen Optokoppler verwenden. Es kommt speziell auf die Zeichen-kombi [L1M](#) an. (1,6mA LED-Strom)

Die Schaltung wurde geändert und entspricht jetzt im Wesentlichen der Standardbeschaltung.

MIDI über USB beherrscht pahlbasic nicht.

Für den Anschluss von MIDI-Breakout-Boards / Platinensynthesizern wird keine zusätzliche Hardware-Schnittstelle benötigt. In der Regel lassen sich solche Boards direkt an die Prozessor-Pins der 2. Seriellen Schnittstelle anschliessen. Boards mit 5V Versorgung evtl. mit Pegelshiftern.

Ich denke da an das [Musical Instrument Shield](#) von Sparkfun oder Boards mit dem [ATSAM2195](#) oder [ATSAM2695](#) oder ähnlich.

Für ein Retro-Projekt ist auch der zukünftige Anschluss von Vintage Yamaha Chips denkbar (das 407ZG-Board hat ein paralleles 16-Bit-Interface...).

## Modbus

Der Modbus ist ein Kommunikationsmittel für eine Steuerungsinfrastruktur, die sich auch für DIY-Projekte eignet. Im Prinzip ginge es mit Basic-Bordmitteln, da es sich um ein serielles Netzwerk handelt. Basis ist ein Feldbus mit RS485-Interfaces. Da es dafür Chips und fertige Module gibt, ist die Realisierung recht einfach. Es können Ausgänge ein- bzw. ausgeschaltet oder abgefragt werden. Eingänge können getestet werden. Mit Modulen für Temperatursensoren können auch Temperaturen erfasst werden. Später kommen evtl. noch weitere Module hinzu möglicherweise auch Displays. Im ersten Schritt werden verschiedene Ein-/Ausgangsmodule mit und ohne Relais unterstützt. Man kann z.B. Fenster- oder Türkontakte überwachen, Rollläden hochfahren, Signalgeber einschalten und mehr. Eine Modell-Eisenbahn mit grossen Abmessungen könnte aber auch ein dankbares 'Opfer' sein.

Das Netz ist kabelgebunden. Am besten eignen sich Fernmeldekabel z.B. 4-adrig IY(ST)Y 2x2x0,6 oder 0,8 (je nach Entfernung) für Spannungsversorgung und Daten. Die Module sollten einen Spannungsregler on Board haben. Dann kann man 9...12V Versorgung durch die Busleitung zu jedem Modul schalten. Der RS485-Bus hat – wie die Wurst – 2 Enden. An beiden Enden wird er mit 120 Ohm-Widerständen abgeschlossen. An einem Ende kann der Basic-Computer angeschlossen sein, ebenso gut kann er aber auch irgendwo in der Mitte sein. Sämtliche Module (und der Basic-Computer) werden elektrisch parallel auf dem Bus angeschlossen. Jeder Busteilnehmer hat die Busanschlüsse: [+12V, Gnd, A, B]. Wobei A und B die symmetrischen Datenleitungen sind, statt 12V können auch andere Spannungen sinnvoll sein.

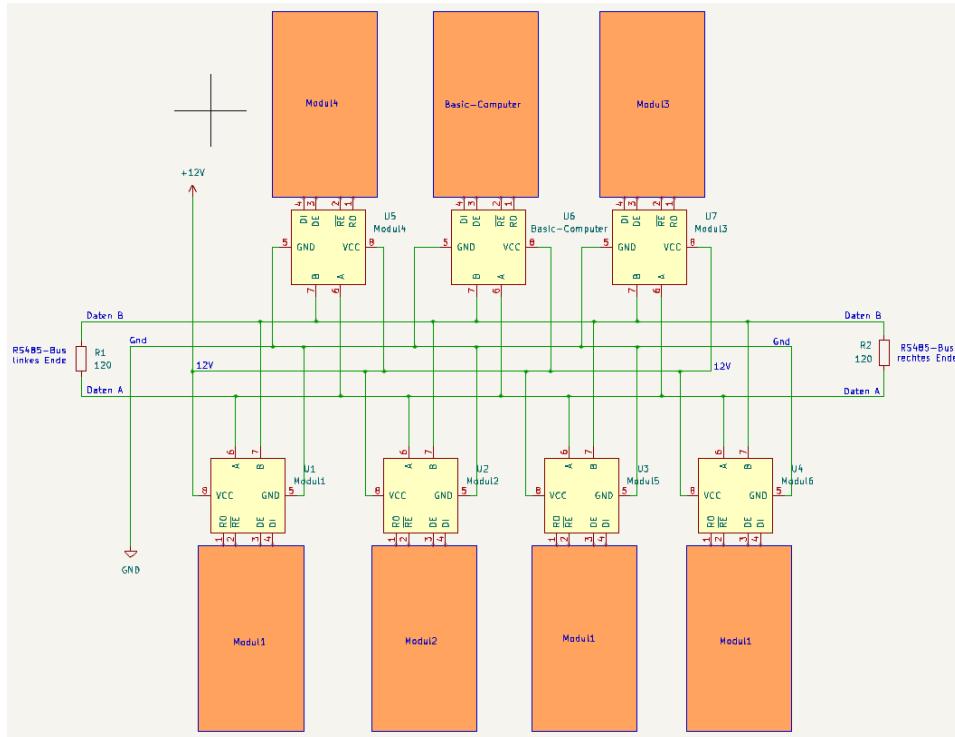
Die Module haben einen einfachen Schutz für den Einsatz auf so einem Feldbus. Auch der Anschluss des Basic-Computers sollte einfache Schutzmassnahmen haben. Es müssen die Bestimmungen des jeweiligen Landes eingehalten werden! Gilt ja für alle elektrischen Installationen, insbesondere wenn Dritte involviert sind. Solange nur Kleinspannungen (z.B. 12V) verwendet werden, ist man auf der sicheren Seite. Der Einbau in Kunststoffgehäuse und der Einsatz zugelassener Komponenten wird empfohlen.

**Sicherheits-relevante** Steuerungen sollte der Basic-Computer nicht vornehmen. Technische Laien holen sich bitte Rat von Fachleuten. Fachleute steuern bitte mit dem Basic-Computer nur elektrische Einrichtungen, die dafür vorgerüstet sind. Ein Garagentor z.B. muss eine eigene Steuerung haben mit geeigneten Eingängen ('auf / zu'), die für den Anschluß externer Geräte vorgesehen sind. Der interne Steuerungsablauf muss von der Garagentoranlage durchgeführt werden (mit allen Sicherheitsvorkehrungen). Der Basic-Computer darf solche Steuerungen nur anstoßen - galvanisch getrennt, rückwirkungsfrei. Wenn der Basic-Computer gestört ist (oder falsch programmiert), muss das Tor trotzdem einwandfrei funktionieren! Bitte **niemals** die Motoren **direkt** ansteuern! Das gilt für alle ähnlich gelagerten Fälle. Den Anschluß darf nur ein Fachmann vornehmen, damit aus dem Garagentor keine Guillotine wird. Sie, als Betreiber sind verantwortlich für Schäden, die aus dem Betrieb der selbstgebauten Anlage erwachsen – gilt aber für alle elektrischen DIY-Projekte. Bei Außeninstallationen Blitzschutz beachten. Ein FI-Schutzschalter ist ein Muß für alle Projekte mit Netzstrom! Anfänger experimentieren besser nur mit Batterien.

Theoretisch könnte man auch eine Alarmanlage programmieren. Diese darf aber nur als **zusätzliche** Informationseinrichtung nicht aber als Gefahrenmeldeanlage betrieben werden. So, damit habe ich meine Pflicht erfüllt, auf mögliche Gefahren hinzuweisen. Nun wieder zurück zur Technik ☺.

Modbus ist ein Master-/Slave-Netzwerk. Der Basic-Computer steuert die Kommunikation als einziger Master. Der Basic-Computer sendet eine Modbus-Zeichensequenz an die Steuermodule standartmäßig mit 9600 Baud und wartet auf die korrekte Bestätigung vom Modul. Das dauert ca. 21ms. z.Z. wird Funktion 05 unterstützt, die einzelne Ausgänge schaltet, sowie Funktion 06, die 16 Ausgänge auf einmal setzt. Hohe Schaltfrequenzen sind daher nicht möglich. Es geht auch eher um sichere Übertragung über längere Entfernung. Rein *theoretisch* sind 1200m möglich. Der von mir eingesetzte SP3485 kann bis zu 32 Module treiben (laut Datenblatt). Es gibt Module mit 32Ausgängen bzw. Eingängen. Ich nutze gerne 16Eingangs-/16Ausgangsmodule, aber auch Module mit 4Relais und 4Eingängen.

Jedes Modul hat eine Modul-Adresse. Bei pahlbasic von 1...32. Vom Prinzip aber 1...255. \*)



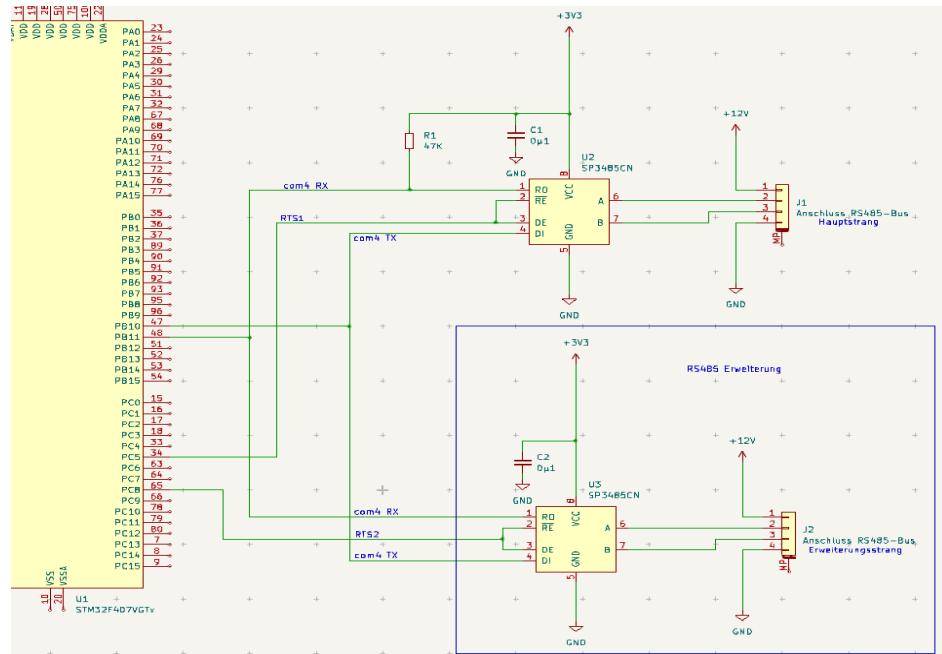
Die Module sind Platinen mit integriertem RS485-Treiber-IC.

Jedes Modul wird mit dem 4-drähtigen Bus verbunden, von dem es auch seine Spannungsversorgung erhält.

Jede Adresse darf nur einmal vorkommen.

Der RS485-Bus als Schaltbild (nur Prinzip)

Den Basic-Computer muss man mit dem Interface ausstatten:



So simpel ist der Anschluss des RS485-Busses. (J1)

Man muss nicht mal selbst etwas zusammenlöten. Es gibt Breakout-boards mit diesem Treiber-IC (U2).

Wie immer muss man auf den richtigen Anschluss von TX und RX achten. Die Bezeichnungen sind manchmal irreführend.

RS485-Bus-Anschluss für den Basic-Computer – hier mit com4 verbunden, R1 nicht weglassen!

Der umrahmte Schaltungsteil wird z.Z. nur auf dem F407ZG-Board unterstützt. Geliefert werden die Module mit eingestellter Adresse 1. Die braucht man ja nur einmal. Deshalb muss man die Adressen ändern. Das Basic bringt das Handwerkszeug dafür mit.

Als erstes die gewünschte Schnittstelle konfigurieren (hier wird com4 verwendet, weil com2 und com3 ja für MIDI und GPS genutzt werden sollen):

`config com=4, 9600, 8, 2, 1`    oder    `config com=4, 9600, 8, 0, 2`

Um die Moduladresse zu ändern, verbindet man ein *einzelnes* Modul mit dem RS485-Anschluss des Basic-Computers (beim F407ZG: Strang1) – 130Ohm Widerstand zwischen A und B nicht vergessen!

#### Folgende Kommandos funktionieren auf den Modulen der Fa. eletechsup:

Ändern der Adresse von Adresse 1 auf 2:

`send #4, 1, 6, 0, 253, 0, 2`    Antwort: 01 06 00 FD 00 02

ist die Adresse unbekannt, an Adresse 255 senden:

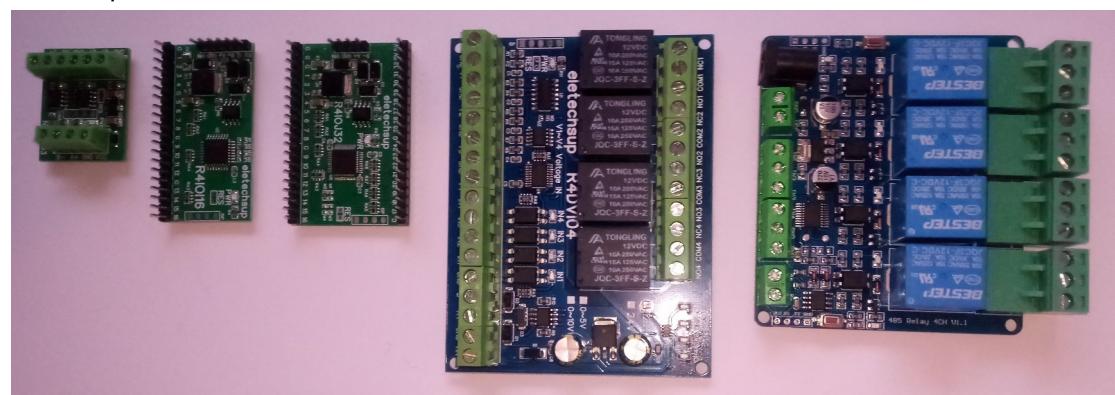
`send #4, 255, 6, 0, 253, 0, 2`    Antwort: FF 06 00 FD 00 02

eine unbekannte Adresse abfragen:

`send #4, 255, 3, 0, 253, 0, 1`    Antwort: FF 03 02 00 03

Die Antworten des Moduls enthalten noch weitere Zeichen (hinter den hier abgedruckten), die aber nur den CR-Check enthalten.

Die von pahlbasic unterstützen Module:



R4IOI16 (16Port-Ein-/Ausgänge) >



< R4IOJ32 (32Port-Ein-/Ausgänge)  
N4DSB03 (2 Sensor-Eingänge für DS18B20)



>



Dann gibt es auch noch die Kombi-Module mit Relaisausgängen, digitalen und analogen Spannungs- und Stromeingängen, von denen z.B. folgende von pahlbasic unterstützt werden:

R4DVIO4 (4 analoge, 4 digitale Eingänge und 4 Relais) siehe rechts. – alle von eletechsup.

Ferner ein Modul der Fa. Shangle Network Technology Co mit 4 Relais und 4 digitalen Eingängen (sehr preisgünstig). Siehe unten. Andere Modbus-Module funktionieren evtl. auch, wenn sie die Funktionen 1,2,3,4,5,6 unterstützen.

Im Prinzip könnte man mit der Anweisung `send#` alle Dinge machen, die man für Modbus braucht. Damit die Handhabung aber etwas weniger umständlich ist, gibt es die Anweisungen `set#`, `reset#` und `get adx#`, `get input#`, `get output#` sowie `get diginp#`, `get relay#`.



(wenn der Leistungsbedarf nicht so hoch ist) oder dezentral mit einem Netzgerät oder Notstromversorgung mit geeigneter Spannung. Der Daten-Bus selbst benötigt nur die 3 Leitungen Gnd, A und B. Der Basic-Computer kann seit Kurzem auch als Modbus-Modul agieren. (Siehe [server](#))

[R4IOI16](#) und [R4IOJ32](#) verstehen die Anweisungen: `set#`, `reset#`, `get input#`, `get diginp#`, `get output#` und `get relay#`. Adressenabfrage und -Einstellung sind oben beschrieben. Ein- und Ausgänge sind im Auslieferungszustand low aktiv. Es können verschiedene Ein-/Ausgangskombinationen bestellt werden.

Die beiden Temperatursensoren (DS18B20) am [N4DSB03](#) werden mit `get ad1#` und `get ad2#` abgefragt, der erhaltene Wert muss durch 10 dividiert werden.

[R4DVIO4](#) versteht die Anweisungen `set#`, `reset#`, `get diginp#`, `get relay#`, `get adx#` mit:  $x=1\ldots 4$ . Die dig. Eingänge müssen aktiv nach low gezogen werden und melden dann ,1‘.

Die Adresse (z.B. [5](#)) vom [4RD7](#) stellt man mit: `send #4, 0, 16, 0, 0, 0, 1, 2, 0, 5` ein. Die Adresse wird abgefragt mit: `send #4, 0, 3, 0, 0, 0, 1`. Für die Adressen-Einstellung/Abfrage darf nur ein Modul angeschlossen sein.

Die Relais setzt / rücksetzt man mit `set#` und `reset#`. Die Eingänge fragt man mit `get diginp#` ab. Die Eingänge (Optokoppler!) melden ohne Beschaltung High-Zustand. Sie werden aktiv nach Gnd geschaltet und melden dann ,0‘. Alle Relais aus: `send #4, 5, 15, 0, 0, 0, 8, 1, 0`. Die Adresse ist hier [5](#), als Schnittstelle wurde Nr.4 angenommen.

\*) Das 407Miniboard hat z.Z. nur einen RTS-Ausgang (RTS1 für die Umwandlung der Datenrichtung) daher auch nur einen RS485-Strang mit max. 32 Modulen (Adressen 1...32).

Das F407ZG-Board hat 3. Man kann daher 3 RS485-Stränge parallelschalten und bis zu 96 Module mit einer Schnittstelle ansprechen. Es darf aber keine doppelten Adressen geben. Strang1 spricht die Adressen 1...32 an, Strang2 die Adressen 33...64 und Strang3 die Adressen 65...96.

Die Module haben alle die 4 Anschlüsse für den RS485-Bus: [+12V](#), [Gnd](#), [A](#), [B](#). Die Spannungsversorgung der Module kann mit der Bus-Spannung erfolgen

## Vom Basic nutzbare Hardware-Interrupts

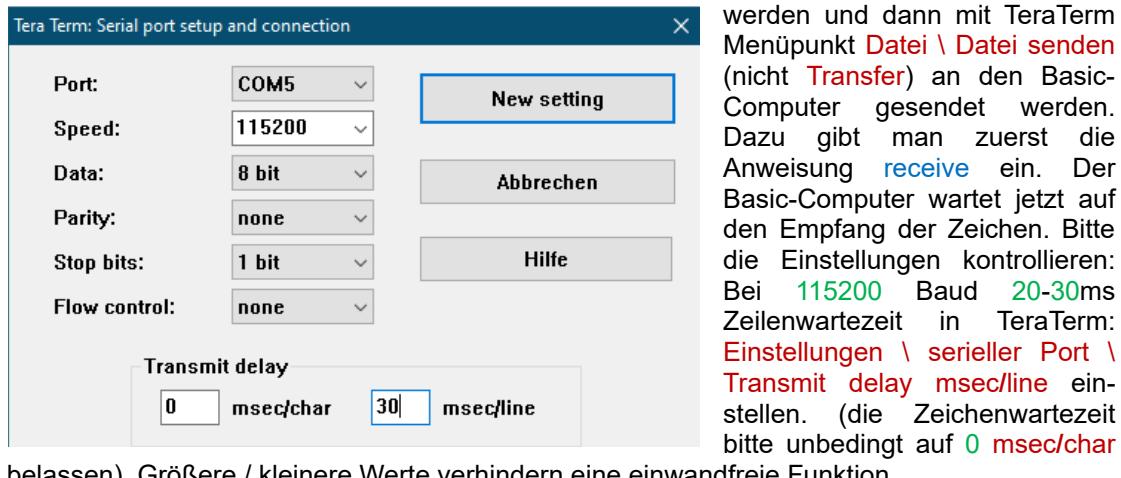
Nr.	Auslöser	on-Name	Priorität *)
00	Ext. Interrupt 0	diginp 1	80
01	Ext. Interrupt 1	diginp 2	81
02	Ext. Interrupt 2	diginp 3	82
03	Ext. Interrupt 3	diginp 4	83
04	Ext. Interrupt 4	diginp 5	84
05	Timer3 Interrupt	time	36
06	USART1-Interrupt	com 1 / Konsole / seriell1	44
07	Software Interrupt	err	---
08	ADC1-Watchdog	ad1 ...ad4	25
09	Timer11 Interrupt	milli1	33
10	USART4-Interrupt	com 2 / midi / seriell2	59
11	USART6-Interrupt	com 3 / gps / seriell3	78
12	Timer11 Interrupt	milli2	33
13	Timer11 Interrupt	milli3	33
14	Ext. Interrupt 14	frei	49
15	Ext. Interrupt 15	touch	49
16	USB global Interrupt	com 5 / Konsole / Keyboard	74
17	USART3-Interrupt	com 4 / seriell4	46
18	RTC Alarm A	clock 1	48
19	RTC Alarm B	clock 2	48
20	Timer10 Interrupt	impuls 1	32
21	Timer10 Interrupt	impuls 2	32
22	Timer10 Interrupt	impuls 3	32
23	Timer10 Interrupt	impuls 4	32
24	Timer11 Interrupt	milli4	33

Prinzipiell können alle on-Interrupts gleichzeitig aktiv sein (mit on Ereignis start gestartet). Da Interrupts aber Prozeduren sind, teilen sie sich die Ressourcen mit den im Hauptprogramm eingesetzten Prozeduren und Funktionen. Es können zurzeit nur 10 Tasks quasi gleichzeitig arbeiten (Das Hauptprogramm ist Nr.1). Interrupts unterbrechen in der Regel nur das Hauptprogramm, höher priorisierte Interrupts können aber auch niedere unterbrechen. Nach Fertigstellung dieser Portierung (wenn der max. Ressourcenverbrauch feststeht), wird dieser Wert noch angepasst. Prozeduren (auch Interrupts) und Funktionen starten jeweils eine komplette Programm-Umgebung und benötigen (anders als Compiler-Funktionen) rund 1,5 – 2KByte RAM sowie etwa 20µs zum Starten und auch zum Beenden, d.h. max. 25000 Prozeduraufrufe pro Sekunde.

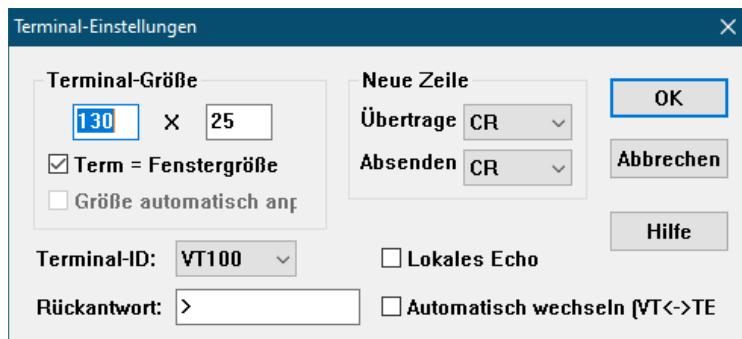
\*) Je höher der Wert, desto niedriger die Priorität

## Datei senden

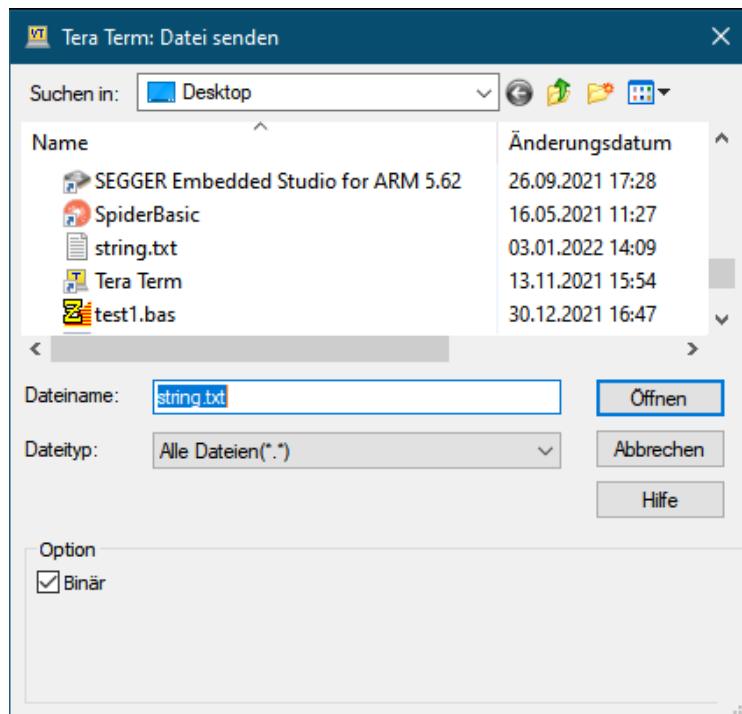
Basic-Programm-Text-Dateien können auch mit einem Texteditor (Editor, Notepad...) erstellt werden und dann mit TeraTerm



werden und dann mit TeraTerm Menüpunkt **Datei \ Datei senden** (nicht **Transfer**) an den Basic-Computer gesendet werden. Dazu gibt man zuerst die Anweisung `receive` ein. Der Basic-Computer wartet jetzt auf den Empfang der Zeichen. Bitte die Einstellungen kontrollieren: Bei 115200 Baud 20-30ms Zeilenwartezeit in TeraTerm: Einstellungen \ serieller Port \ Transmit delay msec/line einstellen. (die Zeichenwartezeit bitte unbedingt auf 0 msec/char belassen). Größere / kleinere Werte verhindern eine einwandfreie Funktion.

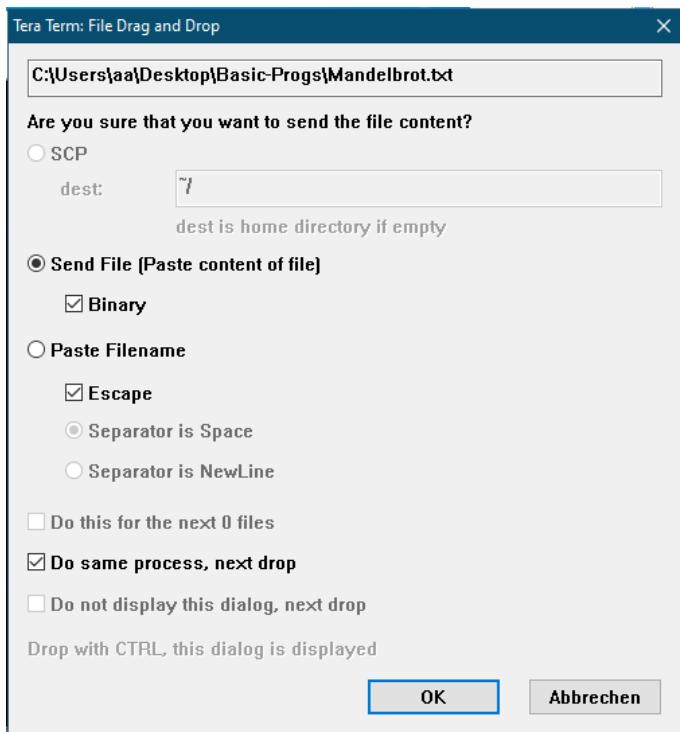


Damit TeraTerm die erfolgreiche Übernahme jeder Programm-Zeile erkennt, muss unter Einstellungen \ Terminal-Einstellungen \ Rückantwort das Prompt-Zeichen > angegeben werden.



Datei unter "Datei senden" öffnen, Option "Binär" aktivieren. Nach erfolgreicher Übertragung gibt der Basic-Computer eine entsprechende Meldung aus. Sind Fehler im Programm-Text, wird die betroffene Zeile nicht übernommen. In diesem Fall werden eine Fehlermeldung und ein Ausdruck der fehlerhaften Zeile auf der Konsole ausgegeben.

Eine alternative Möglichkeit zu **Datei senden** nach Starten von `receive` ist: Drag and drop der Textdatei auf das Terminalfenster. Dann öffnet sich ein Dialogfenster:



Binary aktivieren und OK klicken.

Ich wünsche allen, die den Basic Computer nachbauen, genauso viel Spaß, wie ich bei der Entwicklung hatte.

Mit Anregungen bitte nicht sparen.

## Haftungsausschluss

**Die Verwendung von pahlbasic for Arm geschieht auf eigenes Risiko! Der Autor haftet nicht für Schäden, die durch die Verwendung von pahlbasic for ARM oder Anwendung dieser Anleitung geschehen – auch nicht für Fehler und Irrtümer.**

Anregungen, Kommentare usw. kann man [hier](#) auch anonym loswerden.