Enhanced BASIC language reference By Lee Davison

Version 2.2c/2.3b

Erweiterungen für c't-BASIC siehe unter **c't-BASIC Extensions**
Entfallene Befehle/Funktionen sind rot gekennzeichnet, geänderte grün.
Befehle ab BASIC 2.3
Memory Map siehe PDF **ct-BASIC-MemoryMap**

Numbers

Numbers may range from zero to plus or minus 1.70141173x10^38 and will have an accuracy of just under 1 part in 1.68 x 10^7.

Numbers can be preceded by a sign, + or -, and are written as a string of numeric digits with or without a decimal point and can also have a positive or negative exponent as a power of 10 multiplier e.g.

-142 96.3 0.25 -136.42E-3 -1.3E7 1

.. are all valid numbers.

Integer numbers, i.e. with no decimal fraction or exponent, can also be in either hexadecimal or binary. Hexadecimal numbers should be preceded by **$** and binary numbers preceded by **%**, e.g.

%101010 -$FFE0 $A0127BD -%10011001 %00001010 $0A

.. again are all valid numbers.

Strings

Strings are any string of printable characters enclosed in a pair of quotation marks. Non printing characters may be converted to single character strings using the CHR$() functions.

"Hello world" "-136.42E-3" "+----+----+" "[Y/n]" "Y"

Are all valid strings.

Variables

Variables of both numeric and string type are available. String variables are distinguished by the **$** suffix. As well as simple variables arrays are also available and these may be either numeric or string and are distinguished by their bracketed indicies after the variable name.

Variable names may be any length but only the first two name characters are significant so BL and BLANK will refer to the same variable. The first character must be one of "A" to "Z" or "a" to "z", following characters may also include numbers. E.g.

A A$ NAME$ x2LIM y colour s1 s2

Variable names are case sensitive so AB, Ab, aB and ab are all separate variables. Variable names may not contain BASIC keywords. Keywords are only valid in upper case so 'PRINTER' is not allowed (it would be interpreted as PRINT ER) but 'printer' is.

Note that spaces in variable names are ignored so 'print e r', 'print er' and 'pri nter' will all be interpreted the same way.

BASIC Keywords

Here is a list of BASIC keywords. They are only valid when entered in upper case as shown and spaces may not be included in them. So GOTO is valid BASIC keyword but GO TO is not.

ABS	AND	ASC	ATN	BIN$	BITCLR	BITSET
BITTST	CALL	CHR$	CLEAR	CONT	COS	DATA
DEC	DEEK	DEF	DIM	DO	DOKE	END
EOR	EXP	FN	FOR	FRE	GET	GOSUB
GOTO	HEX$	IF	INC	INPUT	INT	IRQ
LCASE$	LEFT$	LEN	LET	LIST	LOAD	LOG
LOOP	MAX	MID$	MIN	NEW	NEXT	NMI
NOT	NULL	OFF	ON	OR	PEEK	PI
POKE	POS	PRINT	READ	REM	RESTORE	RETIRQ
RETNMI	RETURN	RIGHT$	RND	RUN	SADD	SAVE
SIN	SGN	SPC(	SQR	STEP	STOP	STR$
SWAP	TAB(	TAN	THEN	TO	TWOPI	UCASE$
UNTIL	USR	VAL	VARPTR	WAIT	WHILE	WIDTH
+	-	*	/	^	<<	>>
>	=	<				

- Anything in upper case is part of the command/function structure and must be present
- Anything in lower case enclosed in < > is to be supplied by the user
- Anything enclosed in [ ] is optional
- Anything enclosed in { } and separated by | characters are multi choice options
- Any items followed by an ellipsis, ... , may be repeated any number of times
- Any punctuation and symbols, except those above, are part of the structure and must be included

var	is a valid variable name
var$	is a valid string variable name
var()	is a valid array name
var$()	is a valid string array name

expression	is any expression returning a result
expression$	is any expression returning a string result

addr is an integer in the range +/- 16777215 that will be wrapped to the range 0 to

	65535
b	is a byte value 0 to 255
n	is an integer in the range 0 to 63999
w	is an integer in the range -32768 to 32767
i	is a +ve integer value
r	is real number
+r	is a +ve value real number (0 is considered +ve)
$	is a string literal

BASIC Commands

BITCLR <addr>,<b> Clears bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error.

BITSET <addr>,<b> Sets bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error.

CALL <addr> Calls a user subroutine at address <addr>. No values are passed or returned and so this is much faster than using USR()

DATA [{r|$}[,{r|$}]...] Defines a constant or series of constants. Real constants are held as strings in program memory and can be read as numeric values or string values. String constants may contain spaces but if they need to contain commas then they must be enclosed in quotes.

DEC <var>[,var]... Decrement variables. The variables listed will have their values decremented by one. Trying to decrement a string variable will give a type mismatch error. DEC A is much faster than doing A=A-1 and DEC A,A is slightly faster than doing A=A-2.

DEF FN <name>(<var>) = <statement> Defines <statement> as function <name>. <name> can be any valid numeric variable name of one or more characters. <var> must be a simple variable and is used to pass a numeric argument into the function. Note that the value of <var> will be unchanged if it is used in the function so <var> should be considered to be a local variable name.

DIM <var[$](i1[,i2[,in]...])>[,var[$](i1[,i2[,in]...])]... Dimension arrays. Creates arrays of either string or numeric variables. The arrays can have one or more dimensions. The lower limit of any dimension is always zero and the upper limit is i. If you do not explicitly dimension an array then it's number of dimensions will be set when you first access it and the upper bound will be set to 10 for each dimension.

DO Marks the beginning of a DO .. LOOP loop (See LOOP). No parameters. This command can be nested like FOR .. NEXT or GOSUB .. RETURN.

DOKE <addr,w> Writes the word value w into the addresses addr and addr+1, the lower byte of w is in addr. Note if addr = 65535 ($FFFF) then the high byte will be written to address zero.

END Terminates program execution and returns control to the command line (direct mode). END may be placed anywhere in a program, it does not have to be on the last line, and there may be any number, including none, of ENDs in total.

FN<name>(<expression>) See DEF.

FOR <var> = <expression> TO <expression> [STEP expression] Assigns a variable to a loop counter and optionally sets the start value, the end value and the step size. If STEP expression is omitted then a default step size of +1 will be assumed.

**GET <var[$]>** Gets a key, if there is one, from the input device. If there is no key waiting then var will be set to 0 and var$ will return a null string "". GET does not halt and execution will continue.

GOSUB <n> Call a subroutine at line n. Program execution is diverted to line n but the calling point is remembered. Upon encountering a RETURN statement program execution will continue with the next statement (line) after the GOSUB.

GOTO <n> Continue execution from line number n

IF <expression>[relation expression]

THEN<{{statement|n}|{GOTO|GOSUB}n}> Evaluates expression. If the result of expression is non zero then the statement(s) after the THEN or the GOTO or GOSUB are executed. If the result of expression is zero then execution continues with the next line.

ELSE s.u.

INC <var>[,var]... Increment variables. The variables listed will have their values incremented by one. Trying to increment a string variable will give a type mismatch error. INC A is much faster than doing A=A+1 and INC A,A is slightly faster than doing A=A+2.

INPUT ["$";] <var>[,var] Get a variable, or list of variables from the input stream. A question mark, "?", is always output, after the string if there is one, and if further input is required, i.e. there are more variables in the list than the user entered values, then a double question mark, "??", will be output until enough values have been entered.

There are two possible messages that may appear during the execution of an input statement:

Extra ignored The user has attempted to enter more values than are required. Program execution will continue but the extraneous data entered has been discarded.

Redo from start The user has attempted to enter a string where a number was expected. The reverse never causes an error as numbers are also valid strings.

LET <var> = <expression> Assign the value of expression to var. Both var and expression bust be of the same type. The LET command word is optional and just <var> = <expression> will give exactly the same result. It is only maintained for historical reasons.

LIST [n1][-n2] Lists the entire program held in memory. If n1 is specified then the listing will start from line n1 and run to the end of the program. If -n2 is specified then the listing will terminate after line n2 has been listed. If n1 and -n2 are specified then all the lines from n1 to n2 inclusive will be listed.

LIST can be executed from within a program, first a [CR][LF] is printed and then the specified lines, if any, each terminated with another [CR][LF]. Program execution then continues as normal.

LOAD "<name>.BAS" Load BASIC program <name> from SD card into memory. Starts execution if first line contains REM AUTOSTART.

LOOP This loop repeats forever. With just this command control is passed back to the next command after the corresponding DO.

LOOP [{UNTIL|WHILE} <expression>] Marks the end of a DO .. LOOP loop. There are three possible variations on the LOOP command ..

LOOP UNTIL <expression> This loop will repeat until the value of expression is non zero. Once that occurs execution will continue with the next command after the LOOP UNTIL.

LOOP WHILE <expression> This loop will repeat while the value of expression is non zero. When the value of expression is zero execution will continue with the next command after the LOOP WHILE.

NEW Deletes the current program and all variables from memory.

NEXT [var[,var]...] Increments or decrements a loop variable and checks for the terminating condition. If the terminating condition has been reached then execution continues with the next command, else execution continues with the command after the FOR assignment. See FOR.

NOT <expression> Generates the bitwise NOT of then signed integer value of <expression>.

Note: If n1 does not exist then the list will start from the next line numbered after n1. If n2 does not exist then the listing will stop with the last line numbered before n2.

ON <expression> {GOTO|GOSUB} <n>[,n]... The integer value of expression is calculated and then the nth number after the GOTO or GOSUB is taken (where n is the result of expression). Note that valid results for expression range only from zero to 255. Any result outside this range will cause a Function call error.

POKE <addr,b> Writes the byte value b into the address addr.

PRINT [expression][{;|,}expression]...[{;|,}] Outputs the value of each expressions. If the list of expressions to be output does not end with a comma or a semi-colon, then a carriage return and linefeed is output after the values. Expressions on the line can be separated with either a semi-colon, causing the next expression to follow immediately, or a comma which will advance the output to the next tab stop before continuing to print.

If there are no expressions and no comma or semi-colon after the PRINT statement then a carriage return and linefeed is output. When entering a program line, or immediate statement, PRINT can be abbreviated to ?

READ <var>[,var]... Reads values from DATA statements and assigns them to variables. Trying to read a string literal into a numeric variable will cause a syntax error.

REM Everything following this statement on this program line will be ignored, even colons. REM AUTOSTART as first line starts program automatically after LOAD.

RESTORE [n] Reset the DATA pointer. If n is specified then the pointer will be reset to the beginning of line n else it will be reset to the start of the program. If n is specified but doesn't exist an error will be generated.

RETURN Returns program execution to the next statement (line) after the last GOSUB encountered. See GOSUB. Also returns program execution to the next statement after an interrupt but does not restore the enabled flags.

RUN [n] Begins execution of the program currently in memory at the lowest numbered line. RUN erases all variables and functions, resets FOR .. NEXT, GOSUB .. RETURN and DO ..LOOP states and sets the data pointer to the program start. If n is specified then program execution will start at the specified line number.

SAVE "<name>.BAS" Save BASIC program <name> to SD card.

SPC(<expression>) Prints <expression> spaces. This command is only valid in a PRINT statement.

STEP Sets the step size in a FOR .. NEXT loop. See FOR.

STOP Halts program execution and generates a "Break in line n" message where n is the line in which the STOP was encountered.

SWAP <var[$]>,<var[$]> Swap two variables. The variables listed will have their values exchanged. Both must be of the same type, numeric or string, and either, or both, may be array elements. Trying to swap a numeric and string variable will give a type mismatch error.

TAB(<expression>) Sets the cursor position to <expression>. If the cursor is already beyond that point then the cursor will be left where it is. This command is only valid in a PRINT statement.

THEN See IF.

TO Sets the range in a FOR .. NEXT loop. See FOR.

UNTIL See DO and LOOP.

WAIT <addr,b1>[,b2] Program execution will wait at this point until the value of the location addr exclusive ORed with b2 then ANDed with b1 is non zero. If b2 is not defined then it is assumed to be zero. Note b1 and b2 must both be byte values.

WHILE See DO and LOOP.

WIDTH {b1|,b2|b1,b2} Sets the terminal width and TAB spacing. b1 is the terminal width and b2 is the tab spacing, default is 80 and 14. Width can be zero, for "infinite" terminal width, or from 16 to 255. The tab size is from 2 to width-1 or 127, whichever is smaller.

BASIC Operators

Operators perform mathematical or logical operations on values and return the result. The operation is usually preceded by a variable name and equality sign or is part of an IF .. THEN statement.

+	Add. c = a + b will assign the sum of a and b to c.
-	Subtract. c = a - b will assign the result of a minus b to c.
*	Multiply. c = a * b will assign the product of a and b to c.
/	Divide. c = a / b will assign the result of a divided by b to c.
^	Raise to the power of. c = a ^ b will assign the result of a raised to the power of b to c.
AND	Logical AND. c = a AND b will assign the logical AND of a and b to c
EOR	Logical Exclusive OR. c = a EOR b will assign the logical exclusive OR of a and b to c.
OR	Logical OR. c = a OR b will assign the logical inclusive OR of a and b to c.
<<	Shift left. c = a << b will assign the result of a shifted left by b bits to c.
>>	Shift right. c = a >> b will assign the result of a shifted right by b bits to c.
=	Equals. c = a = b will assign the result of the comparison a = b to c.
>	Greater than. c = a < b will assign the result of the comparison a > b to c.
<	Less than. c = a < b will assign the result of the comparison of a < b to c.

The three comparison operators can be mixed to provide further operators ..

>= or =>	Greater than or equal to.
<= or =<	Less than or equal to.
<> or ><	Not equal to (greater than or less than).
<=> any order	Always true (greater than or equal to or less than).

BASIC Functions

Functions always return a value, be it numeric or string, so are used on the right hand side of the = sign in assignments, on either side of operators and in commands requiring an expression e.g. after PRINT, within expressions, or in other functions.

SGN(<expression>) Returns the sign of <expression>. If the value is +ve SGN returns +1, if the value is -ve then SGN returns -1. If expression=0 then SGN returns 0.

INT(<expression>) Returns the integer of <expression>.

ABS(<expression>) Returns the absolute value of <expression>.

USR(<expression>) Takes the value of <expression> and places it in FAC1 and then calls the USeR routine pointed to by the vector at $0B,$0C. What the routine does with this value is entirely up to the user, it can even be safely ignored if it isn't needed. The routine, after the user code has done an RTS, takes whatever is in FAC1 and returns that. Note it can be either a numeric or string value. If no value needs to be passed or returned then CALL is a better option.

FRE(<expression>) Returns the amount of free program memory. The value of expression is ignored and can be numeric or string.

POS(<expression>) Returns the POSition of the cursor on the terminal line. The value of expression is ignored. Löschkandidat, da wg. CURSOR überflüssig. Nicht mehr benutzen.

SQR(<expression>) Returns the square root of <expression>.

RND(<expression>) Returns a random number in the range 0 to 1. If the value of <expression> is non zero then it will be used as the seed for the returned pseudo random number otherwise the next number in the sequence will be returned.

LOG(<expression>) Returns the natural logarithm (base e) of <expression>.

EXP(<expression>) Returns e^<expression>. Natural antilog.

COS(<expression>) Returns the cosine of the angle <expression> radians.

SIN(<expression>) Returns the sine of the angle <expression> radians.

TAN(<expression>) Returns the tangent of the angle <expression> radians.

ATN(<expression>) Returns, in radians, the arctangent of <expression>.

PEEK(<addr>) Returns the byte value of <addr>.

DEEK(<addr>) Returns the word value of <addr> and addr+1 as an integer in the range -32768 to 32767. Addr holds the word low byte.

SADD(<{var$|var$()}>) Returns the address of var$ or var$(). This returns a pointer to the actual string in memory not the descriptor. If you want the pointer to the descriptor use VARPTR instead.

LEN(<expression$>) Returns the length of <expression$>.

STR$(<expression>) Returns the result of <expression> as a string.

VAL(<expression$>) Returns the value of <expression$>.

ASC(<expression$>) Returns the ASCII value of the first character of <expression$>.

LCASE$(<expression$>) Returns <expression$> with all the alpha characters in lower case.

UCASE$(<expression$>) Returns <expression$> with all the alpha characters in upper case.

CHR$(<b>) Returns single character string of character <b>.

HEX$(<expression>[,b]) Returns <expression> as a hex string. If b is omitted, or if b = 0, then the string is returned with all leading zeroes removed and is of variable length. If b is set, permissible values range from 1 to 6, then a string of length b will be returned. The result is always unsigned and calling this function with expression > 2^24-1 or b > 6 will cause a function call error.

BIN$(<expression>[,b]) Returns <expression> as a binary string. If b is omitted, or if b = 0, then the string is returned with all leading zeroes removed and is of variable length. If b is set, permissible values range from 1 to 24, then a string of length b will be returned. The result is always unsigned and calling this function with expression > 2^24-1 or b > 24 will cause a function call error.

BITTST(<addr>,<b>) Tests bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. Returns zero if the bit was zero, returns -1 if the bit was 1.

MAX(<expression>[,<expression>]...) Returns the maximum value from a list of numeric expressions. There must be at least one expression but the upper limit is dictated by the line length. Each expression is evaluated in turn and the largest of them returned.

MIN(<expression>[,<expression>]...) Returns the minimum value from a list of numeric expressions. There must be at least one expression but the upper limit is dictated by the line length. Each expression is evaluated in turn and the smallest of them returned.

PI Returns the value of pi as 3.14159274 (closest floating value).

TWOPI Returns the value of 2*pi as 6.28318548 (closest floating value).

VARPTR(<var[$]>) Returns a pointer to the variable memory space. If the variable is numeric, or a numeric array element, then VARPTR returns the pointer to the packed value of that variable in memory. If the variable is a string, or a string array element, then VARPTR returns a pointer to the descriptor for that string. If you want the pointer to the string itself use SADD instead.

LEFT$(<expression$,b>) Returns the leftmost b characters of <expression$>.

RIGHT$(<expression$,b>) Returns the rightmost b characters of <expression$>

MID$(<expression$,b1>[,b2]) Returns the substring string from character b1 of expression$ of length b2. The characters of expression$ are numbered from 1 starting with the leftmost. If b2 is omitted then all the characters from b1 to the end of the string are returned.

BASIC Error Messages

These all occur from time to time and, if the error occurred while executing a program, will be followed by "in line " where is the number of the line in which the error occurred.

NEXT without FOR Error

NEXT has been encountered and no matching FOR could be found.

Syntax Error

Just generally worng. 8^)=

RETURN without GOSUB Error

RETURN has been encountered and no matching GOSUB could be found.

Out of DATA Error

A READ has tried to read data beyond the last item. Usually because you either mistyped the DATA lines, miscounted the DATA, RESTOREd to the wrong place or just plain forgot to restore.

Function call Error

Some parameter of a function was outside it's limits. E.g. Trying to POKE a value of less than 0 or greater than 255.

Overflow Error

The result of a calculation has exceeded the numerical range of BASIC. This is plus or minus 1.7014117+E38

Out of memory Error

Anything that uses memory can cause this but mostly it's writing and running programmes that does it.

Undefined statement Error

Either a GOTO, GOSUB, RUN or RESTORE was attempted to a line that doesn't exist or the line referred to in an ON <expression> {GOTO|GOSUB} or ON {IRQ|NMI} doesn't exist.

Array bounds Error

An attempt was made to access an element of an array that was outside it's bounding dimensions.

Double dimension Error

An attempt has been made to dimension an already dimensioned array. This could be because the array was accessed previously causing it to be dimensioned by default.

Divide by zero Error

The right hand side of an A/B expression was zero.

Illegal direct Error

An attempt was made to execute a command or function in direct mode which is disallowed in that mode e.g. INPUT or DEF.

Type mismatch Error

An attempt was made to assign a numeric value to a string variable, a string value to a numeric variable or a value of one type was returned when a value of the other type was expexted or an attempt at a relational operation between a string and a number was made.

String too long Error

String lengths can be from zero to 255 characters, more than that and you will see this.

String too complex Error

A string expression caused an overflow on the descriptor stack. Try splitting the expression into smaller pieces.

Can't continue Error

Execution can't be continued because either the program execution ended because an error occurred, NEW or CLEAR have been executed since the program was interrupted or the program has been edited.

Timeout Error

(neu) LABINP oder *LABREQ* wurde wegen ausbleibender Antwort nach 0,25s abgebrochen.

Undefined function Error

FN <var> was called but not found.

LOOP without DO Error

LOOP has been encountered and no matching DO could be found.

c't-BASIC Extensions

von Carsten Meyer, cm@ctmagazin.de

DIR zur Anzeige des SD-Karteninhalts

LOAD "<name>.<ext>" Laden eines (tokenisierten) Basic-Programms von SD-Karte, z.B. *LOAD "CLOCK.BAS"*, anschließend BASIC-Warmstart (mit Autostart, wenn erste Programmzeile mit *REM AUTOSTART* beginnt).

LOAD <string_expr>|"<"<name>.<ext>",<startadr> lädt beliebige Daten an die Speicherstelle <*startadr*>. Es wird kein BASIC-Warmstart ausgeführt, kann daher auch in Programmen verwendet werden. Für Dateinamen sind Stringvariablen zulässig. Achtung: Es werden immer nur ganze 256-Byte-Pages geladen. Beispiel: *LOAD"PLOTLIB.BIN",$8000*

Zum Abspeichern von Binärdateien siehe Beispielprogramm *BINSAVE.BAS*.

SAVE "<name>.<ext>" Speichern eines (tokenisierten) BASIC-Programms auf SD-Karte

MACRO <string_expr>|"<name>.<ext>" Schickt Textdatei (z.B. auch mit BASIC-Zeilen) an den Interpreter, als wäre sie auf dem PS/2-Keyboard eingegeben worden, oder führt ein auf der SD-Karte vorhandenes LabScript aus, wenn die <ext> "INI" lautet. Ein solches INI-File kann auch Binärdateien in den BASIC-Programmspeicher laden oder speichern (AutoIncrement-Befehle); außerdem besteht mit den neuen TSR-Befehl der Firmware 2.5 die Möglichkeit, in einem LabScript kurze Strings (z.B. BASIC-Befehle) als "Keyboard-Eingabe" an den Interpreter zu schicken. Beispiel: *MACRO "LISTING.MAC"* lädt BASIC-Zeilen nach, die das (vorhandene) Programm auf SD-Karte als Text speichern.

COLOR <expr> Auswahl der Textfarbe, 0..7 normal, 8..15 invers, z.B. *COLOR 15*

CURSOR <x>, <y> Positionierung des Text-Cursors für formatierte Ausgabe. 0,0=links oben, 79,39=rechts unten.

CURSOR ON | OFF Schaltet Underline-Cursor an oder aus, etwa für Textbildaufbau, wenn keine Eingabe verlangt wird. Ansonsten (ON, default) „wandert der Cursor mit“.

LABREQ <mainch_expr>,<subch_expr>,<var> Abfrage eines beliebigen c't-Lab-Moduls an der mit *LABBUS* ausgewählten Schnittstelle (entspricht etwa "0:VAL 3?"). <mainch_expr> und <subch_expr> müssen im Bereich 0..255 liegen (Byte!). Die Rückgabe des abgefragten Messwerts/Registerinhalts erfolgt in <var>, z.B. *LABREQ 0, SUBCH, WERT*. <var> kann auch ein String sein. Textantworten (Fehlermeldungen in []) werden nicht übernommen, hierfür ggf. *LABOUT / LABINP* verwenden. Aufgetretene Kommunikationsfehler führen nicht zum Abbruch, auch der Variableninhalt wird dann nicht verändert. Kommunikationsfehler (z.B. Timeout) können mit LABERR abgefragt werden.

LABSET <mainch_expr>, <subch_expr>, <value_expr> Befehl an ein beliebiges c't-Lab-Modul an der mit LABBUS ausgewählten Schnittstelle, (entspricht etwa "0:VAL 3=1.234!"). <mainch_expr> und <subch_expr> müssen im Bereich 0..255 liegen (Byte!). Es erfolgt keine Rückgabe, z.B. *LABSET 3, SUBCH, WERT*

LABBUS <busnr_expr>[,<bdsel>] Auswahl des Kommunikationskanals. *LABBUS 0* = intern an FPGA-Modul, *1* = OptoBusCore 1 (fest eingestellt auf 38400 Bd), *2* = OptoBusCore 2 bzw. RS-232 auf CORERAM-Platine. Baudrate von *LABBUS 1* und *2* einstellbar mit optionalem zweiten Parameter 0 bis 7: 0=38400, 1=19200, 2=9600 usw. bis 7=300 Baud. *LABBUS* leert UART-Input-Buffer. Ohne zweiten Parameter wird 38400 Bd eingestellt.

LABCLR Entfallen mit #2.1a, stattdessen LABBUS verwenden.

LABOUT <string_expr> Wie *PRINT*, hier aber direkte Ausgabe an eingestellten *LABBUS*, z.B. *LABOUT "0:OPT 20=0"*. Insbesondere sinnvoll für SubCh oberhalb 255 oder alphanumerische Parameter. Nimmt wie *PRINT* auch zusammengesetzte Ausdrücke an, die es vorher berechnet, z.B. *LABOUT MC;":OPT 20=";V+2*

LABINP <string|var> Wie *INPUT*, nur dass die Eingabe vom eingestellten *LABBUS* erfolgt. Es wird die komplette Modulantwort bis zum CR zurückgegeben. Sinnvoll für alphanumerische Antworten, z.B. Filenamen oder Fehlermeldungen, oder c't-Lab-fremde Geräte mit serieller Ansteuerung. Liefern letztere nur eine reine Ziffernfolge (einschl. +, -, E und Punkt), kann statt einer Stringvariablen auch eine normale Variable verwendet werden. *LABINP* setzt keinen Befehl selbst ab, dies hat mit *LABOUT* zu geschehen.

LABERR Funktion, enthält letzte Fehlernummer des Moduls oder Codes für Timeout/Kommunikationsfehler. Wenn kein Fehler aufgetreten ist, liefert *LABERR* den Wert Null. Siehe auch Programm LABINP.BAS. Beispiel: *LABREQ MC,SC,VAL : IF LABERR THEN...*

RENUMBER <von>-<bis>,<neueStartZeilennummer> BASIC-Programm neu nummerieren. Parameter sind optional, zulässig sind also z.B. *RENUMBER | RENUMBER 10-,100 | RENUMBER -2000 | RENUMBER -2000,100*. Achtung: Der Befehl nummeriert *GOTO*s und *GOSUB*-Zeilenangaben nicht um!

ELSE <statements> Ergänzung für IF..THEN/GOTO/GOSUB Konstrukte. Muss in einer eigenen Zeile stehen und einer Zeile mit IF..THEN folgen. Statements hinter ELSE werden ausgeführt, wenn vorangegangenes IF nicht erfüllt war. Siehe Beispielprogramm „IFELSE.BAS“. Geschachtelte ELSEs sind nicht zulässig bzw. es wird immer das Ergebnis des **zuletzt** ermittelten IF-Ergebnisses genommen, also aufpassen bei IF..GOSUB-Konstrukten.

BANK <0..7> schaltet 8 BASIC-Speicherbereiche um (Bankswitch), so dass 8 BASIC-Programme mit je max. 43 KByte Länge gleichzeitig im SRAM gehalten werden können. Nach Umschalten werden Variablen neu initialisiert. *LOAD*- und *SAVE*-Befehle beziehen sich immer auf die aktuelle Bank. Sinnvoll, um zum Beispiel Teile eines Listings in ein anderes BASIC-Programm zu übernehmen (*LIST <Von>-<Bis>*, dann Bankswitch und gewünschte Zeilen mit Shift-ENTER übernehmen).

WINDOW POSITION <sx>,<sy> TO <ex>,<ey> setzt Textfenstergröße von Startkoordinaten *<sx>,<sy>* bis Endkoordinaten *<ex>,<ey>* (bei VGA 0..79 für x, 0..39 für y zulässig). Siehe Programm „WINTEST.BAS“. Zeichenausgaben mit *PRINT* oder *INPUT* erfolgen nur noch innerhalb dieses Fensters. Auch geeignet, um kleinere Ausgabefenster wie bei LCD-Version (40x20) auf VGA (80x40) zu simulieren.

WINDOW CURSOR <x>,<y> relative Positionierung des Schreibcursors innerhalb des mit *WINDOW POSITION* gesetzten Fensters. Nach Neustart oder nach *WINDOW OFF* gleichbedeutend mit *CURSOR <x>,<y>*.

WINDOW OFF setzt Fenstergröße wieder auf gesamten Bildschirm zurück.

WINDOW CLEAR Inhalt des mit *WINDOW POSITION* gesetzten Fensters wird gelöscht. Nach Neustart oder nach *WINDOW OFF* gleichbedeutend mit *PRINT CHR$(12)*.

WINDOW BORDER <rand> mit Wert „1“ als *<rand>* wird das mit *WINDOW POSITION* gesetzte Fenster einfach umrandet, mit „2“ doppelt. „0“ löscht die Umrandung. Farbe ist die mit COLOR gewählte Textfarbe.

FRAME CLEAR Inhalt des Grafikfensters löschen. Hintergrund/Gitterraster bleibt wie vorher.

FRAME GRID <x>,<y>, **FRAME WIDTH <x>,<y>**, **FRAME POSITION <x>,<y>** Grafikfenster-Attribute: Gitterraster-Abstand, Höhe/Breite, Position (linke obere Ecke) gemessen ab Sync-Impuls (nicht mehr im Bild sichtbar)

FRAME COLOR <0..15> Farbe des Grafikfenster-Hintergrunds. 0..7 ohne, 8..15 mit Raster.

FRAME WAIT Synchronisation mit Frame-Aufbau für flackerfreie Anzeige. Wartet bis VGA-Zeile nach dem Grafik-FRAME erreicht ist. Auch zur Verlangsamung von Programmen (max. 1/60 Sekunde Wartezeit).

PLOT CIRCLE <radius>[,<segmbits>] Zeichnet einen Kreis/Kreissegmente mit letzten *PLOT MOVE/POINT/LINE/RECT*-Zielkoordinaten als Mittelpunkt, siehe Beispiel CIRCLE.BAS. Zielkoordinaten bleiben bestehen.

PLOT MACRO <inst>,<wert> - Messinstrument-Makro (eine Art „Sprite“) <inst> (derzeit 0 bis 3) mit angezeigtem Wert (0 bis 100) plotten. Zur Verfügung stehen Zeigerinstrument (0), vertikaler (1) und horizontaler (2) Balken und „Oszilloskop“ (3). Siehe Beispiel INSTR.BAS

PLOT MOVE <x>,<y>, **PLOT POINT <x>,<y>**, **PLOT LINE <x>,<y>**, **PLOT RECT <x>,<y>**, **PLOT COLOR <0..7>** Zeichen-Befehle, gelten immer ab letztem Zielpunkt bzw. *PLOT MOVE*. Punkt *0,0* ist links oben, *255,191* rechts unten. Farben wie bei Text-*COLOR* (ohne invers).

PLOT PRINT <ausdruck> Zahlen (als Vektorgrafik) zeichnen statt Textausgabe, funktioniert wie normales PRINT, kann aber außer "E-+." keine Buchstaben, siehe Beispiel INSTR.BAS

PLOT WAVE <adr>,<wert> Wellenform auf <adr> (Page mit 256 Bytes) anzeigen und um 1 nach links verschieben, neuen <wert> am Ende eintagen, siehe Beispiel WAVE.BAS. Als Datenspeicher empfiehlt sich der freie Bereich ab $B000 bis $B3FF, z.B. *PLOT WAVE $B000,Y*.

PLOT SIZE <x>,<y> - Skalierung für *PLOT PRINT* und *PLOT MACRO* (0,0 bis 2,2, 0 = normale, 1=doppelte, 2=vierfache Größe)

PLOT LIST <adr> Eigene Vektorenliste (je vier Bytes X,Y,Farbe,Befehl) auf Adresse <adr> zeichnen. Tabelle kann Callbacks für die Annahme von weiteren Parametern enthalten (so funktioniert auch *PLOT MACRO* intern). Siehe Sourcecode *plotlib_cm.asm*

PRINT CHR$(2) schaltet die Bildschirmausgabe auf den mit *LABBUS* ausgewählten Kommunikationskanal um. Kann genutzt werden, um beispielsweise das BASIC-Listing auf der RS-232-Schnittstelle auszugeben oder in Klartext auf SD-Karte zu speichern (siehe Macro LISTING.MAC). ASCII-Wert 2 (STX) wird bereits als Start-Zeichen an die mit *LABBUS* eingestellte Schnittstelle ausgegeben. Siehe auch alternatives *WINDOW CLEAR*.

PRINT CHR$(3) schaltet die Bildschirmausgabe auf den Bildschirm zurück. ASCII-Wert 3 (ETX) wird noch als Ende-Zeichen an die vorher benutzte Schnittstelle ausgegeben.

CALL -9 Monitor aufrufen, Ende mit X-Taste

CALL -12 Terminal aufrufen, Ende mit ESC-Taste

Achtung: Viele dieser Befehle arbeiten nur ab **FPGA-Firmware Version 2.6**, die um einige neue Befehle (u.a. Routinen zum Speichern von Textdaten auf SD-Karte, Echtzeituhr) erweitert wurde. Bitte beachten Sie auch die Beispielprogramme im Verzeichnis "sdcard" der ct-BASIC-Sourcen (Repository, ZIP in Ordner entpacken!). Im Ordner "ct-BASIC" befinden sich die Assembler-Sourcen für BASIC-Interpreter, Monitor und Terminal-Programm (Hauptdatei „main_cm.asm“), so dass Sie diese auch nach eigenen Vorstellungen erweitern können (6502-Kenntnisse erforderlich, von Microsoft-/Applesoft-BASIC hilfreich). Hier sind auch sämtliche Hardware-Adressen enthalten.

Tastenbelegung

Die Tastenbelegung ist für eine deutsche Tastatur ausgelegt, Umlaute sind allerdings nicht möglich. Eckige/geschweifte Klammern wurden auf "ÖÄ" gelegt. F10 bis F12 dienen (wie der *LABBUS*-BASIC-Befehl) zur manuellen Auswahl des Kommunikationskanals, etwa im Terminal-Programm oder im Monitor (F10=*LABBUS 0*, F11=*LABBUS 1* usw.). Im BASIC zusätzlich:

F1 LIST
F2 DIR
TAB kopiere Zeichen unter Cursor in Eingangspuffer
Shift-ENTER übernehme gesamte (ggf. editierte) Zeile, in der sich der Cursor befindet
INSERT Zeichen-Einfügemodus ein/aus, automatisch aus beim nächsten ENTER
Ctrl-C bricht laufendes BASIC-Programm ab
DRUCK/PRNTSCR löscht den Text-Bildschirm und das Grafikfenster (*PRINT CHR$(12)* löscht nur den Text).
PAUSE führt ein Reset des 6502-Cores aus, danach gelangt man per Menü in den Interpreter (Kaltstart, Warmstart löscht vorhandenes Programm nicht), zum Terminal oder in den 6502-Monitor.