TinyBasic

www.tinybasic.de

programming manual

version 2.4

TinyBasic

Programming Manual

Version 2.4
February 2009

altenburg

CONTENTS

APPENDIX

CHAPTER 1 Introduction

TINYBASIC

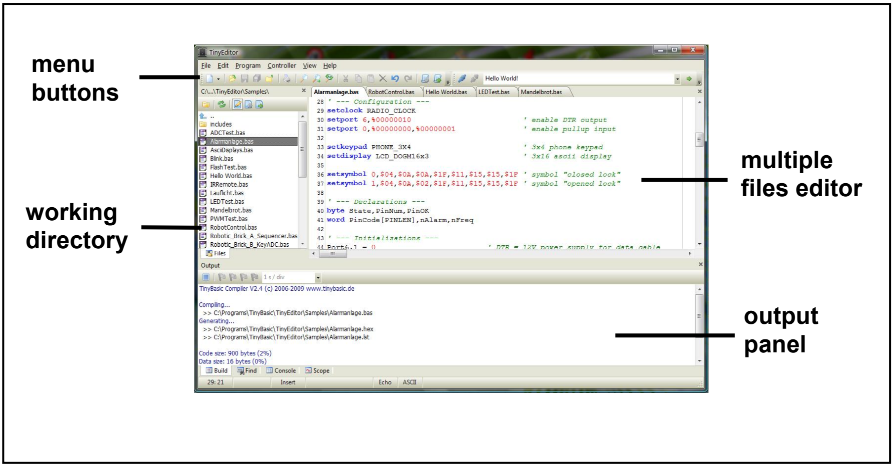
TinyBasic is a very simple and easy to learn programming language designed for embedded microcontroller systems. Especially low volume applications and single solutions are suitable for TinyBasic.

Common applications are:

- home automation,
- temperature control,
- data logging and display,
- measurement and diagnostics,
- GSM- or modem communication.

EDITOR

TinyBasic comes with a full featured source code editor. The editor supports multiple files in workbook mode. It also has a working directory view and, of course, an output window for compiler messages.



```
' --- Configuration ---
setclock RADIO_CLOCK
setport 6,%00000010                          ' enable DTR output
setport 0,%00000000,%00000001                ' enable pullup input

setkeypad PHONE_3X4                          ' 3x4 phone keypad
setdisplay LCD_DOGM16x3                      ' 3x16 ascii display

setsymbol 0,$04,$0A,$0A,$1F,$11,$15,$15,$1F ' symbol "closed lock"
setsymbol 1,$04,$0A,$02,$1F,$11,$15,$15,$1F ' symbol "opened lock"

' --- Declarations ---
byte State,PinNum,PinOK
word PinCode[PINLEN],nAlarm,nFreq

' --- Initializations ---
Port6.1 = 0                          ' DTR = 12V power supply for data cable
```

```
TinyBasic Compiler V2.4 (c) 2006-2009 www.tinybasic.de

Compiling...
  >> C:\Programs\TinyBasic\TinyEditor\Samples\Alarmanlage.bas
Generating...
  >> C:\Programs\TinyBasic\TinyEditor\Samples\Alarmanlage.hex
  >> C:\Programs\TinyBasic\TinyEditor\Samples\Alarmanlage.lst

Code size: 900 bytes (2%)
Data size: 16 bytes (0%)
```

The working directory view as well as the output panel can be hidden by selecting the menu item 'View/Output Panel' or 'View/Working Directory' respectively. This allows the use of the entire screen for source code editing.

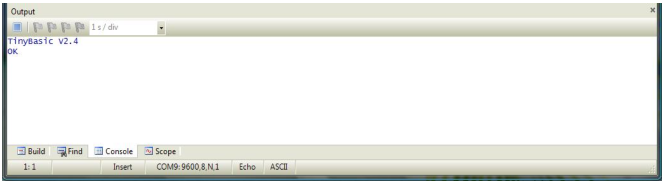
COMPILER

The currently shown source file can be compiled by pressing F9 or by selecting the appropriate menu item 'Program/Compile'. In case of errors in any of the compiled files (a source file can include further files by using the #include directive) the output panel shows a message. Error messages can be double clicked to open the file and directly go to the appropriate source line.

If no errors were found the compiler generates two more files within the directory of the source file. These are a listfile (*.lst) and the codefile (*.hex) in Intel-Hex-File format. The listfile is needed in case of a runtime error which is reported according to a specific address. This address can be found in the listfile together with the appropriate source line.

DOWNLOAD

Prior to the first download, it is recommended to ensure a proper serial connection to the target controller. In order to do so, connect the host PC with the microcontroller first, then select 'Controller/Connect' from the menu and finally power up the controller or press the reset button.



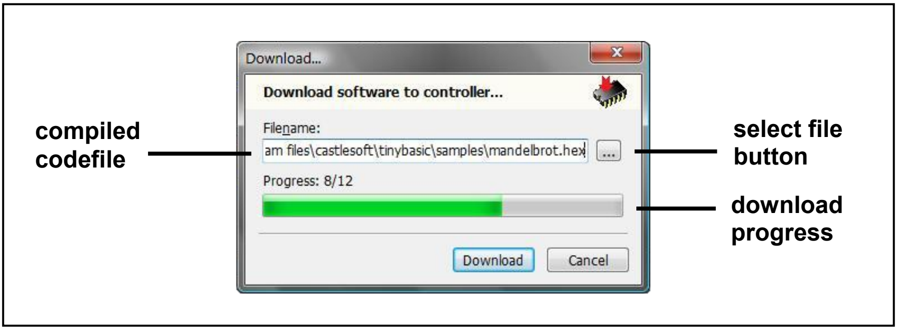
```
TinyBasic V2.4
OK
```

1:1 | Insert | COM9: 9600,8,N,1 | Echo | ASCII

serial settings

The target always sends a message string: **`TinyBasic Vx.x OK`** after coming out of the reset. If this message string is shown in the console window, then the connection will be established.

By pressing F10 or via the menu item 'Program/Download...' the previously compiled Intel-Hex-File (*.hex) can be transferred to the target controller.



Download software to controller...

Filename: am files\castlesoft\tinybasic\samples\mandelbrot.hex

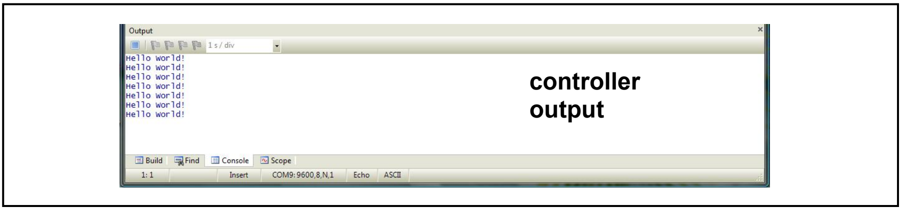
Progress: 8/12

The download itself is performed at 9600 baud, 8 data bits, 1 stop bit and no parity (9600,8,N,1). This is a very common setting for UART communication.

Normally, the name of the compiled Intel-Hex-File is already copied to the 'Filename' edit box within the download dialog, when it opens. In order to select a different file for the download, there is a select file button on the right side of the 'Filename' edit box.

CONSOLE

To test an application program the console window automatically appears after the download has been finished. All outputs serially sent from the controller are shown in this window. Normally, these are all PRINT statements not redirected to other output devices like the display.



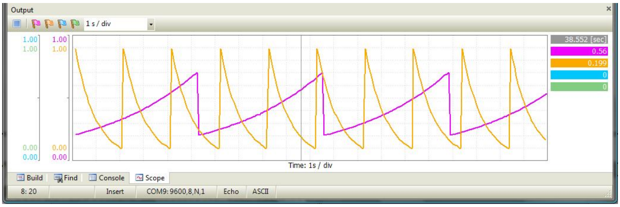
```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

1:1 Insert COM9:9600,8,N,1 Echo ASCII

The console window can also write to a logfile. To begin a logging session select ‘Logging…’ from the context menu and choose an existing logfile or enter a new file name. All subsequent messages from the controller will be written to the selected logfile. To stop logging select the ‘Logging…’ menu again.

SCOPE

The TinyEditor includes a virtual 4 channel oscilloscope, which can be used to scope analog as well as digital values from a TinyBasic application.



To send values to one of the four channels of the scope the following PRINT syntax schould be used:

```
PRINT ”#<channel>”,<value>,…
```

Where <channel> is in the range 0..3 and <value> is a variable or expression. Note the final comma to suppress the <CR/LF>. Of course more than one channel can be set with a single PRINT statement.

Samples:

```
PRINT ”#0”,x,
PRINT ”#0”,x,”#1”,y,”#2”,z,
PRINT ”#0”,x,”#1”,sin(x),”#2”,cos(x),
PRINT ”#0”,x,”#1”,sin(x),”#2”,y,”#3”,sin(y),
```

CHAPTER 2
Preprocessor

#TARGET <target>

The #TARGET directive tells the compiler for which target it has to compile. Even though the basic commands are identical for all targets there are different memory sizes and limitations concerning the maximum array length. The following targets are supported by now: TINYBRICK8, TINYBRICK16 and TINYDISPLAY.
Note: A program compiled for TINYBRICK8 will also run on a TINYBRICK16 or a TINYDISPLAY, but not necessarily vice versa.

#INCLUDE ”<filename>”

An application can consist of more than one file. In this case a file can include another file by using the #INCLUDE directive. The given file name can be absolute (e.g. ”c:\programs\test.bas”) or relative to the location of the file (e.g. ”\includes\test.bas”).

#DEFINE <macro>(<paramlist>) <replacement>
#DEFINE <macro> <replacement>
#DEFINE <macro>
#UNDEF <macro>

Macros are textual replacements within the source code. They are very useful to define named constants, e.g.: #DEFINE Timeout 1000. Every occurrence of Timeout will then be replaced by 1000, e.g.: PAUSE Timeout will be replaced to PAUSE 1000.
It is also possible to replace default variable names with more sophisticated identifiers. If, for instance, a motor is connected to Port2.0 then the program would be easier to read by using the following replacement: #DEFINE Motor Port2.0, because it is then

possible to write Motor = 1 resp. Motor = 0 to switch the motor on or off.

Macros can also have parameters. A single parameter or even a list of parameters separated by commas is given in parentheses following the macro name (without any space). Each occurrence of a parameter within the replacement will then be replaced by its actual parameter given whenever the macro is evaluated.

```
#DEFINE VALUE(x)     ″Value = ″,x
#DEFINE VALUES(x,y) ″Value1 = ″,x,″ Value2 = ″,y

PRINT VALUE(1+1)     ` PRINT ″Value = ″,1+1
PRINT VALUE(ADC0)    ` PRINT ″Value = ″,ADC0
PRINT VALUES(1,2)    ` PRINT ″Value1 = ″,1,″ Value2 = ″,2
```

The directive #UNDEF can be used to remove a macro definition.

#IFDEF <macro>
#IFNDEF <macro>
#ELSE
#ENDIF

The directives #IF(N)DEF, #ELSE and #ENDIF are used for conditional compilation. Different versions of the same source code can be compiled depending on the definition of switches (simple macros without a replacement text).

```
#DEFINE GRAPHIC_LCD          ` compile for graphic display


#IFDEF GRAPHIC_LCD
  SETDISPLAY LCD_DOGM128x64
#ELSE
  SETDISPLAY LCD_DOGM16x3
#ENDIF
```

CHAPTER 3
Variables and Types

TYPES

TinyBasic supports several integral data types as well as a floating point type.

CHAR	0..255	□	(1 byte)
BYTE	0..255	□	(1 byte)
WORD	0..65535	□□	(2 bytes)
INTEGER	-32768..32767	□□	(2 bytes)
LONG	-2147483648..2147483647	□□□□	(4 bytes)
FLOAT	approx. 6 digits precision	□□□□	(4 bytes)

In order to save variable memory it is recommended to use always the smallest possible data type. The TinyBasic variable memory is limited to 1024 bytes (*TinyBrick8*) or 2048 bytes (*TinyBrick16/ TinyDisplay*). Floating point operations take more time than integral operations. Therefore floating point types should be preserved for numeric calculations only.

VARIABLES

Variables are used to store temporary values during program execution. Up to 52 different variables can be used inside a TinyBasic program (26 with TinyBrick8). This seems to be very restrictive, but it is enough for many applications. Variables must be declared prior to their first use. Although this is different to many other basic implementations, it helps to keep the needed memory amount as small as possible. Each variable must have an unique name – its identifier. A variable identifier must start with

```
BYTE    bValue,bTemp
WORD    wValue,wMask
INTEGER nPos,nLength,i,j
LONG    lCounter,lNumbers
FLOAT   fDistance,fScaleFactor
```

a letter or an underscore followed by letters or even numbers. `MyVariable`, `Temp`, `Temp_1`, `Value`, `_help` or even `__12345` are valid variable identifiers. There is no limitation to the length of identifiers and they are case-insensitive. It is helpful to begin each identifier with a prefix according to its type. This makes the source code more readable, but it is not required.

ARRAYS

Arrays are collections of single variables in a vector addressable manner. They will be declared by simply adding a number in brackets to a simple variable. It is allowed to declare array variables together with simple variables in the same line.

```
BYTE    b[8]
CHAR    strText[20]
FLOAT   fSum[10],fAverage
```

A special data type is a character array – it will be treated as a string. A string is terminated by a null character. Therefore a string declaration needs to be at least one character longer than the longest text it will contain during program execution.

b[0]	**b[1]**	**b[2]**	**b[3]**	**b[4]**	**b[5]**	**b[6]**	**b[7]**

Note: BYTE b[8] declares b[0]..b[7], b[8] is out of range!

Arrays can be initialized with a single assignment. This is quite comfortable and, of course, the fastest way to do so. The line: b = [0,1,2,3,4,5,6,7] initializes the elements b[0] with 0, b[1] with 1 and so on. Character arrays can also be initialized with text constants: strText = "Hello World!", which is a short form of: strText = ["H","e","l","l","o"," ","W","o","r","l","d","!",0]. Note: It is not required to initialize all elements of an array.

There are some special statements which support other ways to initialize variables from DATA lines or from the controller’s flash memory. It is also possible to store variables into the controller's flash memory. These statements will be discussed now.

DATA <item>,<item>,..

The DATA statement provides a convenient way to insert constants into programs. One or more DATA statements can be grouped together to form a whole set of data elements. The data can be read at any time and as often as needed. DATA lines can contain a list of simple constant values as well as string constants and can be placed anywhere in the program. According to the variable type, which is used to read the data, a suitable conversion will be performed. As it is also allowed to use expressions instead of simple values any calculation can be performed to scale or normalize values.

```
Table:
  DATA 1,2,3,1.0,2.0,3.0,"Hello World!"
  DATA 2.1*f,2.3*f,2.5*f,2.7*f
```

READ <var>,<var>,..

To read the values stored in DATA lines the READ statement must be used. Following the keyword READ a single variable or a list of variables is required. The READ statement fetches as many items as needed to fill up each variable. The values fetched will be converted to fit the variable type. In case an array variable is given to the READ statement each field will be fetched. This results in a very short way to initialize an array. This is especially useful for character arrays, which can be initialized with strings, too.

```
RESTORE Const
READ x,y                'declared as FLOAT x,y
READ a                  'declared as BYTE a[5]
READ c                  'declared as CHAR c[15]

Const:
  DATA 1.0,2.0
  DATA 1,2,3,4,5
  DATA "Hello World!"
```

RESTORE <label>

```
DO
  RESTORE Block1          'use data block 1

  READ a                  'declared as BYTE a[5]
  READ c                  'declared as CHAR c[15]

  PRINT "a=",a," c=",c

  RESTORE Block2          'use data block 2

  READ a                  'declared as BYTE a[5]
  READ c                  'declared as CHAR c[15]

  PRINT "a=",a," c=",c
LOOP

Block1:
  DATA 1,2,3,4,5
  DATA "Hello World!"

Block2:
  DATA 6,7,8,9,10
  DATA "Crazy World!"
```

A RESTORE statement is required to specify the DATA set to be used by the next READ. Note: The RESTORE statement is always required even prior to the first use of READ.

STORE <addr>,<varlist>

The STORE statement saves a given list of variables into the flash memory of the controller. Each variable can be of any type - even arrays are supported. The address value can be in the range $2400..$2BFF (*TinyBrick8*) or in the range $F000..$FFFF

```
BYTE b
WORD w
LONG l

STORE $F000,b,w,l   'store variables
```

(*TinyBrick16/TinyDisplay*). These memory ranges are devided into two separated blocks. Block A starts at address $2400 (*TinyBrick8*) resp. $F000 (*TinyBrick16/TinyDisplay*) while Block B starts at address $2800 (*TinyBrick8*) resp. $F800 (*TinyBrick16/TinyDisplay*). If the STORE statement is called with one of these two block addresses the according block will be erased first.

LOAD <addr>,<varlist>

The LOAD statement loads values, previously stored into the flash memory of the controller, into a given list of variables. As with the STORE statement the variable list can contain any data type even arrays. A combination of STORE and LOAD statements allows to save and restore profile settings or data loggings.

```
BYTE b
WORD w
LONG l

LOAD $F000,b,w,l    'reload a variable list
```

CHAPTER 4
Maths and Expressions

CONSTANTS

The following number systems and their notations are supported for numeric constants:

Number system	Notation
binary	%01010101010101010
hexadecimal	$0123456789ABCDEF
decimal	0123456789
floating point	0.123456789

Note: There is no scientific notation for floating point values.

EXPRESSIONS

Expressions are combinations of numeric values with their operations. Common operations are addition, subtraction, multiplication and division. TinyBasic also supports shift-, compare- and logical operations. The following table gives an overview of all supported operations together with their according priority level (operator precedence).

Operator	Priority level
`() unary + - not`	1 (highest)
`* / << >> mod and`	2
`+ - or xor`	3
`< <= > >= <>`	4 (lowest)

The order in which expressions are evaluated depends on the priority of the used operations. An operation with a higher priority level is performed prior to an operation with a lower priority. This is exactly the same as with mathematical equations: 2+3*4=14. By using parentheses the default order can be changed: (2+3)*4=20.

Embedded control applications often need to modify or test values on a bit-level. By applying the logical operations: NOT, AND, OR or XOR to integral values any possible bit manipulation can be achieved.

NOT	AND	OR	XOR
0 → 1 1 → 0	0 0 → 0 0 1 → 0 1 0 → 0 1 1 → 1	0 0 → 0 0 1 → 1 1 0 → 1 1 1 → 1	0 0 → 0 0 1 → 1 1 0 → 1 1 1 → 0

Additionally there are two bit-shift operations: shift left (<<) and shift right (>>). As all integral types are internally represented as 32bit signed values, a shift operation also shifts the sign bit (bit number 31). Note: To preserve negative values a right shifted sign bit will be set afterwards, but if it is shifted to the left it will be lost. This behavior is also known as arithmetic shift. A shift right operation can be seen as a division by two, while a shift left operation can be seen as a multiplication by two.

Finally, there is a modulo division operator MOD, which calculates the rest of a division, e.g. 10 MOD 3 is 1.

CONDITIONS

Control flow statements like: IF, ELSIF and WHILE expect a conditional expression – shorter a condition. A condition results either to null, which is seen as FALSE, or to any other value, which is seen as TRUE. In addition to numeric expressions a condition can also contain relational operations. Therefore, a simple condition can be: IF x<10 THEN..., but it is also possible and, of course, a very common practice to build more complex conditions by using logical operations: IF (x>0) and (x<10) THEN....

How does it work inside? The relational operations (<, <=, >, >=, <>) return to 0 or –1, if the relation is fulfilled. As mentioned earlier, any nonzero value is seen as TRUE. Furthermore, the value –1 has set all of its bits, so bitwise logical operations with other subexpressions are always valid.

Note: To combine relations with logical operations it is required to use parentheses, because the relational operators are of lower priority.

MATHS

Integral data types are preferred for embedded applications, because of their accuracy and faster speed. TinyBasic also supports a 32bit floating point type with single precision (approx. 6 digits). Attention must be paid, when combining integral and floating point types in one expression. TinyBasic expressions are calculated integral as long as possible. Therefore, a subexpression consisting of integral types only will be calculated with LONG type. As soon as one operand is of floating point type the calculation will be performed with FLOAT type.

Note: A division 3/4 results to 0, but 3.0/4 or 3/4.0 results to 0.75. This behavior is very common to higher programming languages, even though other basic implementations always apply floating point division. It is simple to force a subexpression to be calculated with float type by multiplication with 1.0, but it is more difficult to cast a float value down to a long type while maintaining its accuracy.

CHAPTER 5
Control Flow

IF..THEN..

Conditional execution of a sequence of statements can be achieved by using the IF..THEN.. statement. In case the condition following the keyword IF results to zero all statements following the keyword THEN in this line will be skipped over. The statements are separated by colons.

```
IF a>10 THEN a=0 : b=1
IF a>10 THEN a=0 : GOTO lab1
IF a>10 THEN GOSUB sub1 : a=0
```

IF..THEN
...
[ELSIF..THEN]
...
[ELSE]
...
ENDIF

Conditional execution of two or more alternative code sequences can be achieved by using the IF..ENDIF statement. The condition following the keyword IF will be evaluated first. If the result is not zero then the statements following the keyword THEN will be executed. If the first condition results to zero and one or more optional keywords ELSIF are given, all conditions following these keywords will be evaluated consecutively. If any of these conditions results to a non zero

```
IF a>b THEN         'first check a>b
  a=b
ELSIF a>c THEN      'then check a>c
  a=c
ELSE                'else
  a=1
ENDIF
```

value, then the following statements will be executed. If all results of all conditions are zero, then the statements following the optional keyword ELSE will be executed, otherwise all statements between THEN and ENDIF will be skipped.

IF..GOTO <label>
IF..GOSUB <label>
IF..EXIT
IF..RETURN

The IF statement has some special short forms regarding branches. It is allowed to use the keywords EXIT, GOTO, GOSUB and RETURN instead of THEN. This kind of statement is called 'conditional jumps'.

```
IF a>10 RETURN      'conditional return
IF a>10 GOSUB sub2  'conditional gosub
```

DO
...
LOOP

Normally, an endless loop is used in any TinyBasic program – the so called 'main loop'. The fastest way to do so, is to use the DO..LOOP statement. It is possible to leave this loop by using an EXIT statement within the loop.

```
DO                   'the main loop
  GOSUB ReadSerial
  GOSUB SendAnswer
LOOP
```

Note: It is not allowed to jump into or out of any loop in TinyBasic. Always use the keyword EXIT to skip a loop.

WHILE..
...
WEND

The WHILE..WEND statement is a conditional loop. All statements between WHILE and WEND are repetitively executed as long as the condition following the keyword WHILE results in a nonzero value.

```
WHILE a<10          'test condition
  PRINT "a=",a
  a=a+1
WEND                'go to condition
```

FOR..TO..[STEP..]
...
NEXT

A counting loop is represented by the FOR..NEXT statement. In TinyBasic the FOR statement can count all data types including FLOAT values. This is quite comfortable and makes mathematical calculations much easier. Note: The loop counter must be a simple variable. Array indexes are not allowed. The expressions following the keywords TO and STEP are evaluated only once.

```
FOR i=0 TO 100 STEP 2           'iterate up integer
  PRINT "i=",i
NEXT

FOR x=100.0 TO 0.0 STEP -2.0    'iterate down float
  PRINT "x=",x
NEXT
```

If the expression, following the keyword STEP, results in a negative value, the loop will count down, otherwise the loop will count up. The statements between FOR and NEXT will be executed as long as the loop counter value is less or equal to the limit, when counting up or the loop counter value is greater or equal to the limit, when counting down.
Note: It is possible to modify the loop counter variable during execution of the loop.

EXIT

```
FOR i=0 TO 100
  IF i>5 EXIT     'exit loop, if i>5
NEXT
```

To leave a loop statement use the EXIT statement. The EXIT statement can be used within a FOR, WHILE or DO statement. In conjunction with an IF the EXIT statement can also be used as a conditional exit.

GOTO <label>

An unconditional jump to a specific program location can be performed by using the GOTO statement. The jump target can be any label in the program.

```
GOTO L1       'branch to label L1
  ...
L1:
```

Note: Jumps into or out of a loop are not allowed.

GOSUB <label>

A call to a subroutine can be performed by using the GOSUB statement. The subroutine can begin at any label in the program and must be finished by a RETURN statement.

```
GOSUB Welcome 'branch to subroutine
...

Welcome:
  PRINT "Hello World!"
  RETURN      'return to caller
```

Note: Up to 5 nested subroutines are possible.

ON..GOTO <label0>,<label1>,..
ON..GOSUB <label0>,<label1>,..

In conjunction with ON, the GOTO or GOSUB statement extends to a calculated branch. The result of the expression following the keyword ON is taken as an index in the list of labels following the keyword GOTO or GOSUB. The first label will be taken, if the expression results to zero, the second label will be taken, if the expression results to one and so on. If the expression is below zero or above the number of labels in the list no branch will be performed.

```
ON Msg-1 GOSUB Msg1,Msg2,Msg3

Msg1:
  PRINT "Msg 1 received."
  RETURN
Msg2:
  PRINT "Msg 2 received."
  RETURN
Msg3:
  PRINT "Msg 3 received."
  RETURN
```

RETURN

The RETURN statement restores the program execution just behind the end of the last GOSUB or ON..GOSUB call. Each subroutine must finally execute the RETURN statement.

```
main:
  GOSUB sub1 : GOSUB sub2    'note the colon
  GOTO main
sub1:
  PRINT "Sub 1 called."
  RETURN                     'return to caller
sub2:
  PRINT "Sub 2 called."
  RETURN                     'return to caller
```

WAIT <condition>
WAIT <condition>,<timeout>
WAIT <condition>,<timeout>,<label>

The WAIT statement halts the program execution until the given condition results to a nonzero value. In case a timeout value is given the WAIT statement continues if this time is elapsed. If also a label is specified, the WAIT statement branches to that location in case the timeout is elapsed. The maximum timeout value is 65535 milliseconds.

```
receive:
  WAIT Rxd > 5,1000,timeout  'wait max. 1 sec for 5 chars
  GET String,5,CR            'read short string
  PRINT String
  RETURN                     'return to caller
timeout:
  PRINT "Nothing received."
  RETURN                     'return to caller
```

PAUSE <1/1000 sec>

To slow down program execution the PAUSE statement can be used. The given time is measured in milliseconds. The statement PAUSE 1000 delays execution for 1 second. The maximum time value is 65535.

CHAPTER 6
Functions

LO(<word>)

LO() returns the lowest byte of the given word value. For example, LO(257) returns 1, because the internal representation of 257 is $00000101 and therefore the lowest byte is $01.

HI(<word>)

HI() returns the high byte of the given word value. For example, the result of HI(512) is 2, because the hexadecimal value of 512 is $00000200 and the second byte is $02.

MIN(<long>,<long>)
MIN(<float>,<float>)

The MIN() function compares two values and returns the lower one. The values may be of integral or of floating point type. In case that both values are integral then the result is also integral. If at least one value is of floating point type, then the result will also be of floating point type.

MAX(<long>,<long>)
MAX(<float>,<float>)

The MAX() function compares two values and returns the larger one. As with the MIN() function the parameters may be of integral or of floating point types. If both types are integral, then the result will be integral, too.

SIN(<float>)

The SIN() function calculates the sinus value according to the given angle. The angle parameter must be given in radians. The result is a floating point value in the range: -1.0..1.0. Some interesting points are: SIN(0) is 0, so is SIN(PI), SIN(2*PI) and so on.
Note: PI is a predefined constant with the value: 3.1415927.

COS(<float>)

The cosinus function is the counterpart to the sinus function. The parameter must be given in radians. To change values given in degrees to radians the RAD() function can be used.

TAN(<float>)

The tangens function calculates the quotient: sinus/cosinus.

ATN(<float>)

The arcus tangens function is calculated by calling the ATN() function. Arcus tangens is useful for triangulation.

DEG(<float>)
RAD(<float>)

To convert between degrees (0..360°) and radians (0..2*PI) two functions can be used: DEG(), which converts into degrees and RAD, which converts into radians. For example, DEG(PI) is 180, while RAD(180) is PI.

SQR(<float>)

The SQR() function calculates the square root of a given parameter.

EXP(<float>)

The EXP() function calculates the exponentiation of x to the base e, exp(x) = e^x .

LOG(<float>)

The LOG() function calculates the natural logarithm to the base e. This function is the inverse function of EXP() – therefore LOG(EXP(3)) = 3 and EXP(LOG(3)) is also 3.

POW(<long>,<int>)
POW(<float>,<float>)

The POW() function calculates the exponentiation pow(x,y) = x^y. There are two versions of this function. If both parameters are of integral types then the result will also be an integral, otherwise the result is a floating point value.

ABS(<long>)
ABS(<float>)

The ABS() function calculates the absolute (the positive) value to a given parameter. If the parameter is integral, then the result will also be an integral, otherwise the result is a floating point value.

INT(<float>)

The INT() function returns the integral part of a floating point value as an integral type. The function just removes the fractional part from the given value, e.g. INT(3.6) = 3 and INT(-3.6) = -3.
Note: In other basic implementations the function call int(-3.6) would return –4 (the next lower integral value).

ROUND(<float>)

The ROUND() function returns the intergal value next to the given float parameter, e.g. ROUND(-3.4) = -3 while ROUND(-3.5) = -4.

LEN(<stringvar>)

To get the current length of a string variable (declared as an array of char) the LEN() function should be used. A string value stored in a character array is terminated by a trailing zero. The LEN() function counts the characters up to the delimiter (not included) and returns the number.

POS(<stringvar>,<substring>)
POS(<stringvar>,<substring>,<index>)

The POS function finds a substring within a string variable and returns its start position. As with all arrays index counting starts with zero, so the first possible position of the substring is 0. If the substring was not found, then the result value will be –1.
If an additional start index is given, then the search will start at this index of the string variable. This will be especially useful if more than one substring is expected and each should be found in an iteration.
Note: The substring must be a text constant – variables are not allowed.

VAL(<stringvar>)
VAL(<stringvar>,<index>)

As strings can contain numeric values, e.g. ”Pi = 3.1416”, the VAL() function can be used to read such strings and return the numeric value. If no start index is given, the VAL() function will start to read at index 0, otherwise it will start at the given index. Especially, a combination of the VAL() function together with the POS() function allows a flexible string parsing.
Note: The result value returned by the VAL() function is always of floating point type.

POINT(<xpos>,<ypos>)

The POINT() function returns the color value of the pixel given by the two parameters, e.g. POINT(10,20) = 0 when the pixel is cleared.

PEEK(<addr>)

The PEEK() function allows direct memory access. Any address in the range $0000..$FFFF may be used. As a result the byte value at this memory location is returned.

EOF(<file>)

The EOF() function returns -1 (TRUE) if the file specified by the device parameter is read to the end 0 (FALSE) otherwise.

CHAPTER 7

Input and Output

DEVICES

TinyBasic handles standard input and output via device numbers. A device number is like a logical channel. The following device numbers are supported by now:

Device number	Output channel
#0 (optional)	first serial interface (download)
#1	second serial interface
#2	display and keypad
#3	first file handle
#4	second file

Note: In all input and output statements the device number is optional. If no device is specified, the default device (#0) will be selected, which is the first serial interface.

PUT #<device>,<value>
PUT #<device>,<value>,<length>

The PUT statement sends one or more characters to the given device. If an array variable is used as second parameter the number of characters to send can be specified with the optional length parameter. Note: If no length parameter is given only one character will be sent, even if an array is used as value.

GET #<device>,<variable>
GET #<device>,<variable>,<length>
GET #<device>,<variable>,<length>,<delimiter>

The GET statement reads one or more characters from the specified input device to the given variable. If an array variable is given, the

optional length parameter will limit the number of characters to read. If no length is given just one character will be read, even if an array variable is used.
Additionally an optional delimiter can be specified. If either the given number of characters or the delimiter is read then the GET statement will return. The delimiter itself is not placed into the array variable – instead the null character is used.
Note: The GET statement is blocking. If no input characters are available, the GET statement will not return.

INPUT #<device>,<variable>,...
INPUT #<device>,<text>,<variable>,...

Numeric value inputs can be requested by using the INPUT statement. The program is halted until a CR (carriage return) or an ESC (escape) character is received. As it is possible to request more than one variable with a single INPUT statement (e.g. INPUT a,b,c), the different values can be separated by CR or , (comma). If a given variable is an array, one value for each field in the array will be expected. An optional text is sent prior to the input procedure.

PRINT #<device>
PRINT #<device>,<text>,...
PRINT #<device>,<text>,<value>,...
PRINT #<device>,<text>,<format>(<value>),...

The simple PRINT statement sends an optional text or the result of an expression followed by CR (carriage return) and NL (new line) characters to the given device. To suppress the CR, NL characters a trailing , (comma) can be used (e.g. PRINT ”Hello”,). It is, of course, possible, to send multiple text constants as well as multiple expressions in one single PRINT statement (e.g. PRINT ”a=”,a,”b=”,b,”c=”,c) all separated by commas. If an array variable is given all fields of the array will be printed out. This is especially good for debugging purposes. If the array is of char type then a string will be printed out.

Additional specifiers can be used to format the output properly. These format specifiers are discussed in the following sections.

NL
CR

The format specifiers CR and NL are simple constants, which can be used to send the character codes 13 and 10 (decimal) respectively. CR sets the output position back to the beginning of the current line, while NL sets the output cursor to the current position in the next line.

HEX(<value>)

The HEX format specifier forces the value given in parentheses to send with hexadecimal notation. The width of the hex string is a multiple of 2. Hexadecimal values in the range from 0..F are emitted as 00..0F. Values in the range from 100..FFF are emitted as 0100..0FFF and so on. Note: There is no leading ”$” or trailing ”h” character in this notation. This can be easily added to the PRINT statement.

CHR(<value>)

The CHR specifier forces a numeric value given in parentheses to be emitted as its corresponding character code, e.g. PRINT CHR(13) is the same as PRINT CR.

SPC(<number>)
TAB(<position>)

The SPC specifier emits as many spaces as given in parentheses, while the TAB specifier sets the cursor to the position which is given as parameter, if it is not yet behind this position.

USING(<format>,<value>)

To format a numeric value explicitely the USING specifier can be used. The two parameters are a format string and a value which is then formatted according to place holders inserted in the format string. The place holders can consist of a decimal dot and a number of digits (#) in front of and behind.

```
        PRINT USING(”Price ##.## $”,3.5)            ‘ Price  3.50 $
        PRINT USING(”Fuel  ##### lit”,100)          ‘ Fuel   100 lit
```

If no dot is given, floating point values will be truncated – only the integral part will be shown. If a dot is given, integral values will be formatted as floating point values. Unused leading place holders are replaced by spaces, therefore the values appear right aligned.

OPEN <filename> AS #<device>
OPEN <filename> FOR INPUT AS #<device>
OPEN <filename> FOR OUTPUT AS #<device>

Opens a file on a connected SD card. *Under development!*

CLOSE #<device>

Closes a file. *Under development!*

FLUSH #<device>

The FLUSH statement clears the receive buffer of the selected device. The first serial interface (device #0) has a buffer of 32 characters. With the system variable RXD the current number of pending characters in the buffer can be checked. The second serial interface can only hold one pending character.

FIND #<device>,<text>
FIND #<device>,<text>,<timeout>
FIND #<device>,<text>,<timeout>,<label>

The FIND statement can be used to skip characters in an input stream (serial buffer or file) until a match with the given text parameter accures. The text parameter must be a constant. As the statement stops the program execution a timeout parameter can be added. If the given time is elapsed, then the program execution continues. There are two possible ways to handle an error. In case the text is not found the ERR variable is set to 31 (see error codes). This can be evaluated by a following condition statement. The alternative way is to add a jump label as last parameter to the FIND statement. In case the search text could not be found the program execution continues at the given label. The maximum timeout value is 65535 measured in ms.

INITGSM #<device>,<pin>
INITGSM #<device>,<pin>,<initstring>
INITGSM #<device>,<pin>,<initstring>,<time>

To init a connected GSM modem after starting it up this statement should be used. The device parameter is optional and can be one of the two serial interfaces. The second parameter is the 4 digit pin number required by the modem. The statement checks if the pin must be entered. The optional initstring parameter allows a user defined initialization sequence according to the requirements of the specific modem (e.g. to set the internal message storage memory). As a user initialization can take a few seconds the last parameter is the maximum time the statement has to wait for the final OK from the modem. If not specified the default timeout is 2 seconds. After a completed initialization the ERR variable should be checked. If ERR is not zero the initialization should be retried (see error codes).

Note: The INITGSM statement tries to setup the SMS text mode (AT+CMGF=1). If this mode is not supported by the modem (see error codes), the SENDSMS and RECVSMS commands can not be used.

SENDSMS #<device>,<phone>,<text>,...
SENDSMS #<device>,<phone>,<text>,<value>,...
SENDSMS #<device>,<phone>,<text>,<format>(<value>),...

One of the features provided by all GSM modems is the receive and transmit of short text messages, also known as Short Message Service (SMS). The SENDSMS statement handles the complete communication sequence with a connected modem to transmit such a text message. The optional device parameter specifies one of the two serial interfaces. By default the first serial interface (device #0) will be used. The second parameter specifies the destination phone number. This can be a text constant or an character array. Finally a sequence of text constants in combination with numeric values or expressions can be send. The SENDSMS statement excepts the same format options as the PRINT statement does.

Note: As the SMS transmission process can take up to 10 seconds the SENDSMS statement does not wait for the final OK from the modem. If this is needed a FIND statement can be used.

RECVSMS #<device>,<phone>,<message>

To receive text messages from a connected GSM modem the RECVSMS command can be used. The optional device parameter specifies the serial interface to be used. The second parameter must be a character array. The phone number of the sender will be stored to this parameter. Finally a message buffer (also a character array) must be given. The received message text will be stored to this buffer. The maximum length of a SMS message is 160 characters. If the message buffer is shorter, than the message will be truncated to this size. In case an error occurred the ERR variable will be set (see error codes). If no message was received both the phone number as well as the message buffer are empty. This can be check easily with then LEN() function.

Note: A received message will be deleted from the internal message storage area of the modem (AT+CMGD). As this can take several seconds the RECVSMS command does not wait for the final OK from the modem. This can be done with a following FIND statement.

Note: Some modems require a special command (AT+CPMS) to set the internal storage memory for incoming text messages. This setting can be done with the initstring parameter of the INITGSM command.

ERR

Some of the input/output statements assign a dedicated error code to the ERR variable, if an error occured. Otherwise this variable will be cleared to zero. It is also possible to change the variable by a normal assignment. The following error codes can occur:

Error	**Description**
0	OK - no error condition occured
	Fatal errors (program execution stops!)
1	out of data (READ statement)
2	out of range (array index access)
3	out of memory (too many variables)
4	too many nested DO loops
5	too many nested FOR loops
6	too many nested WHILE loops
7	too many nested GOSUB subroutine
8	LOOP without DO
9	NEXT without FOR
10	WEND without WHILE
11	RETURN without GOSUB
	GSM errors (INITGSM,SENDSMS,RECVSMS)
20	GSM_NO_RESPONSE
21	GSM_INVALID_PIN
22	GSM_INIT_FAILED
23	GSM_NO_TEXTMODE
24	GSM_SEND_FAILED
25	GSM_RECV_FAILED
	Filesystem errors (OPEN,PUT,GET,...)
30	FILE_OPEN_ERROR
31	FILE_FIND_ERROR
32	FILE_READ_ERROR
33	FILE_WRITE_ERROR

CHAPTER 8
Date and Time

SETCLOCK <mode>

The SETCLOCK command enables date and time counting in one of the following modes.

Mode	Description
0	Clock is disabled at all
1	SoftClock mode (no external crystal is needed)
2	RealClock mode (external 32.768kHz crystal needed)
3	RadioClock mode (DCF77 radio clock *and* external 32.768kHz crystal needed)

In RadioClock mode the internal clock is updated each time a valid DFC77 signal will be received. If the DCF77 signal is not present the internal clock is driven by the external crystal.

TIME
TIME.HOUR
TIME.MINUTE
TIME.SECOND

TinyBasic uses date and time variables to access the clock. It will depend on the special hardware if both variables are supported, but at least the time structure can always be used. If the variable TIME is read itself, the current day time in seconds is returned. Therefore, at 6:00am the TIME variable holds the value: 6 * 3600 = 21600 seconds, at 12:00am the value is 43200 seconds and so on.
As the TIME variable has read and write access, the realtime clock can also be set by writing the day time in seconds to this variable.

To get the current hour, minute or the current seconds the appropriate identifier must be specified and separated by a .(dot).

These values also have read and write access, e.g. TIME.SECONDS = 0 will clear the seconds.

Note: Care must be taken, when setting the realtime clock by changing hours, minutes and seconds separately. This could lead to an unintended carry over.

DAY

By using the DAY variable the day of the week can be read and also be written. DAY variable is 0 for Sunday, 1 for Monday and so on.

DATE
DATE.DAY
DATE.MONTH
DATE.YEAR

When reading the variable DATE the returned value is the current date calculated in seconds:

YEAR * 31556952 + (MONTH - 1) * 2629746 + (DAY - 1) * 86400.

The reason for that calculation is that an absolute time stamp can easily be generated as: TimeStamp = DATE + TIME, due to the equivalent time base of DATE and TIME. Therefore, a complete date and time information can be packed into a long variable either for sending it over network/radio or for storing it into memory (e.g. data logger application).

As the DATE variable has read and write access it is also possible to set the current date as once by writing the appropriate value.

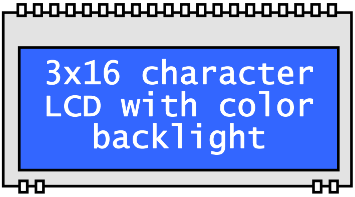
To get the current day, month or year the appropriate identifier must be specified and separated by a .(dot). These values also have read and write access, e.g. DATE.YEAR = 6. Note: The years are counted with two digits only.

Note: The date variable will be available only with TinyBrick16 or TinyDisplay.

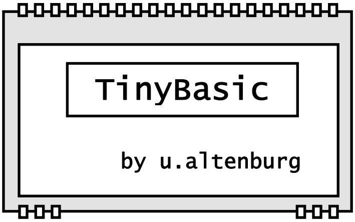
CHAPTER 9
Displays and Graphics

SETDISPLAY <display>

TinyBasic supports different display types ranging from simple 7-segment LED displays with up to 8 digits via 1x8, 2x16 or 3x16 alphanumeric character LCDs up to a fully graphic display with 128x64 pixels.



Electronic Assembly
DOGM-16x Series



DOGM-128x64 Series

The following table shows the supported displays.

Display	Description
0	7-segment displays (driver MAX7219)
1	1x8 characters text display (DOGM-1x8)
2	2x16 characters text display (DOGM-2x16)
3	3x16 characters text display (DOGM-3x16)
4	128x32 pixels graphic display (DOGM-128x32)
5	128x64 pixels graphic display (DOGM-128x64)

SETSYMBOL <charcode>,<data>,...

The alphanumeric text displays support up to 8 user defineable characters/symbols. The command SETSYMBOL expects the character code (0..7) as the first parameter. The following 8 data bytes define the pixels of the 5x8 symbol.

CLS

The CLS command clears the screen and sets the cursor to the home position (0,0 = top left corner). The graphic display screeen will be filled with the current background color.

FONT <font>

The alphanumeric text display has a built-in character set with a fixed size, therefore this command is ignored. The graphic display has one font with two possible sizes – FONT 0 (small font 6x8 pixels) and FONT 1 (large font 12x16 pixels). By default the small font is selected.

COLOR <ink>
COLOR <ink>,<paper>

The COLOR command sets the foreground color and optionally the background color. Valid color values are 0 or 1 (resp. black or white).
Note: This command is ignored by the alphanumeric display.

PLOT <xpos>,<ypos>

The PLOT command sets one single pixel at the location (xpos,ypos) with the current foreground color. The given point is kept as the new cursor location.
Note: This command is ignored by the alphanumeric display.

MOVE <xpos>,<ypos>

The MOVE command sets the cursor to location (xpos,ypos). The values are given in pixels with the graphic display and in characters with the alphanumeric display.

DRAW <xpos>,<ypos>
DRAW <xpos>,<ypos>,<mode>

The DRAW command draws a line with the current color from the current cursor position to the location specified by the parameters xpos and ypos. The last point of the line is set as the new location, therefore multiple DRAW commands can be used to draw a curve. The mode parameter specifies the drawing mode: 0=solid, 1=dotted.
Note: This command is ignored by the alphanumeric display.

FRAME <xpos>,<ypos>,<xrec>,<yrec>
FRAME <xpos>,<ypos>,<xrec>,<yrec>,<fillcolor>

The FRAME command draws a rectangle from the top left corner (xpos,ypos) to the point (xrec,yrec) with the current foreground color. If the fillcolor value is specified the rectangle will be filled, otherwise only a frame will be drawn.
Note: This command is ignored by the alphanumeric display.

SCROLL <xpos>,<ypos>,<xrec>,<yrec>,<dx>
SCROLL <xpos>,<ypos>,<xrec>,<yrec>,<dx>,<dy>

With the SCROLL command a rectangular area of the display can be scrolled. The top left corner of the area is point (xpos,ypos) while the bottom right corner is point (xrec,yrec). The parameter dx is the scroll width of the area contents. If dx is a positive value the area will be scrolled to the right, while negative values scroll the area to the left. The block, which is empty after the scrolling will be filled with the background color.
Note: This command is ignored by the alphanumeric display.

CIRCLE <xpos>,<ypos>,<rad>
CIRCLE <xpos>,<ypos>,<xrad>,<yrad>
CIRCLE <xpos>,<ypos>,<xrad>,<yrad>,<fillcolor>

With the CIRCLE command a circle or an ellipsis can be drawn. The point (xpos,ypos) is the midpoint of the circle or ellipse. The simple rad parameter is the radius of a circle, while the two parameters

xrad and yrad specify the dimension of an ellipse. If a fillcolor is specified then the circle or ellipsis will be filled.
Note: This command is ignored by the alphanumeric display.

PICTURE <xpos>,<ypos>,<picture>
PICTURE <xpos>,<ypos>,<picture>,<mode>

With the PICTURE command a symbol or an icon can be drawn on the graphic screen at the position given by xpos and ypos. The icon data is stored in a byte array given as the picture parameter. The first two bytes in the picture array are the width and the height of the icon. Width and Height have to be multiples of 8.
Note: This command is ignored by the alphanumeric display.

```
  BYTE Icon[18]

Main:
  READ Icon : PICTURE 75,40,Icon    'draw icon
  GOTO Main

Symbols:
  DATA 16,8                         'width,height
  DATA %00000110,%00000000          '     **
  DATA %00001001,%00000000          '    *  *
  DATA %00010000,%10000000          '   *    *
  DATA %00100000,%01000000          '  *      *
  DATA %01100000,%01100000          ' **      **
  DATA %00100000,%01000000          '  *      *
  DATA %00111111,%11000000          '  ********
  DATA %00000000,%00000000          '
```

BARGRAPH <xpos>,<ypos>,<xbar>,<ybar>,<value>
BARGRAPH <xpos>,<ypos>,<xbar>,<ybar>,<value>,<mode>

The BARGRAPH command allows to draw a sizeable bar. The position and maximum size of the bar is defined by the points (xpos,ypos) and (xbar,ybar). The actual size is set by the value parameter in the range 0..100 percent. Depending on the relative position of the two corners and according to the given mode the bar can be drawn in horizontal (mode=0) or vertical (mode=1) direction.
Note: This command is ignored by the alphanumeric display.

CHAPTER 10
Networking

SETNETWORK <net>
SETNETWORK <net>,<baudrate>

TinyBrick modules as well as TinyDisplays can be connected to simple networks. All modules are equipped with a RS485 transceiver. Alternatively, the second serial interface (device #1) can be used in combination with other transceivers, e.g. wireless easy radio modules.

The SETNETWORK command enables transfer of messages with up to 32 bytes in length. The first parameter is the network type – this should be set to null for RS485. The optional baudrate parameter can be specified – otherwise the default baudrate will be 9600 baud.

SEND <id>,<var>,…

To send a message via the network the SEND command can be used. Basically, all messages have an ID (1..255). This ID must be given as the first parameter of the SEND command. A single variable or even a list of variables of any type can be added. If an array variable is used then all fields of the array will be send. Take care, that the size of all variables does not exceed 32 bytes.
Note: By default, there is no address information – all messages are send broadcast.

MSG

How can a receiver detect an incomming message? By reading the MSG variable. The MSG variable holds the ID of the last received message. According to this ID a dedicated RECV command can be used to read the contents of the message. If the receiver is not

interested in some messages these message can be discarded by setting MSG back to null.
Note: MSG can be set to any value, but only a null value will delete a received message.

RECV <var>,…

According to the ID of the last received message a dedicated RECV command should be used to read out the contents of the message. Therefore, the ID can also be seen as a kind of type describtion.

```
  BYTE  nSender                ’ sender
  WORD  wWindSpeed             ’ wind speed
  FLOAT fTemperature           ’ temperature

Main:
  DO                           ’ main loop
    IF Msg THEN                ’ any message
      ON Msg - 1 GOSUB OnTempMsg,OnWindMsg
      Msg = 0                  ’ delete message
    ENDIF
  LOOP

OnTempMsg:                     ’ recv temperature…
  RECV nSender,fTemperature ’ read byte,float
  RETURN

OnWindMsg:                     ’ recv wind speed…
  RECV nSender,wWindSpeed      ’ read byte,word
  RETURN
```

CHAPTER 11
Hardware

SETPORT <port>,<config>
SETPORT <port>,<config>,<pull-up>

All ports of the controller are initialized as inputs with no pullups by default. To change the configuration of a port the SETPORT command can be used. The first parameter is the port number and may be in the range 0..9 (depending on the controller). The configuration of each port pin is set by the second parameter. This parameter is a byte value, where each bit belongs to a port pin direction: SETPORT 2, %01000001 initializes Port2.0 and Port2.6 as output. Note: The most right bit is pin 0 while the most left bit is pin 7.
Each pin initialized as input (config bit set to 0) can also have a pull-up resistor. To activate the resistor set the appropriate bit in the third parameter: SETPORT 2, %01000001, %00000010.
Note: Depending on the controller not each pin can have a pull-up resistor or some pull-ups must be activated together.

PORT*n*
PORT*n.b*

Access to the controller ports is provided by the variables PORT0..PORT9 (depending on the controller). Each port can be set or read back. Additionally each port pin can be set and read by appending the appropriate pin number to the port variable separated by .(dot). The value of a port variable may be 0..255 (8 bits) while the value of a port pin may only be 0..1.

Note: If a value is assigned to a pin variable, only the lowest bit will be written to the port pin. Therefore, the expression PORT2.0 = NOT PORT2.0 is valid, even though the NOT operator inverts all bits.

Note: As TinyBasic only supports PORT0..9 the port 10 of the M16C microcontroller used for TinyBrick16 and TinyDisplay is mapped to the variable PORT0.

SETCOM <com>,<baudrate>

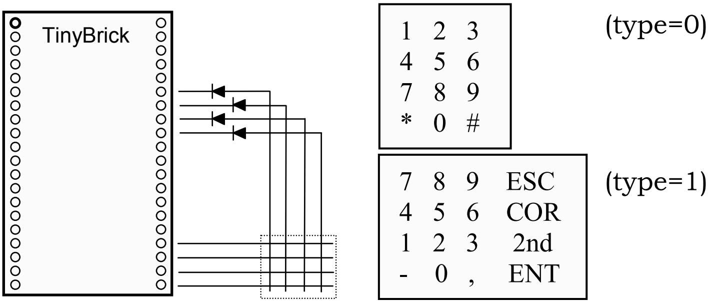
There are two supported serial interfaces #0 and #1. The first serial interface (download interface, device #0) is initialized by default with 9600 baud, 8 data bits, no parity and one stop bit (9600,8,N,1). To change the baudrate of this serial interface the command SETCOM 0,<baud> can be used at any time in the program. The baudrate parameters can be: 300, 600, 1200, 2400, 4800, 9600, 19200 and 38400 with TinyBrick8 and also 57600 and 115200 with TinyBrick16 and TinyDisplay.
The second serial interface is not initialized by default, but can be initialized and set to a specific baudrate by using the command SETCOM 1,<baud>.

RXD

The variable RXD holds the number of pending characters in the receive buffer of device #0. Up to 32 characters will be buffered. To clear this buffer use the FLUSH #0 command.
Note: The second serial interface (device #1) is not buffered.

SETKEYPAD <type>

With the SETKEYPAD command one of two possible keypads can be



selected. Type=0 sets a phone keypad and type=1 sets a numeric keypad with four funktion keys: ESC=escape, COR=correct, 2nd=second function and ENT=enter.

KEYCODE

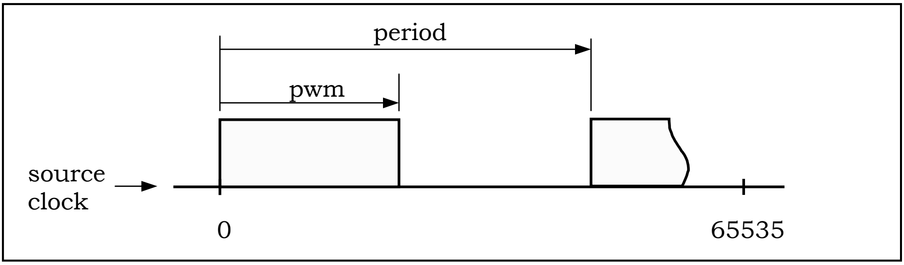
The KEYCODE variable holds the actual pressed key or keys. KEYCODE is a 16bit variable – each bit represents one key.

IRCODE

TinyBasic can receive infrared signals from a television remote control. The remote control must use the RC5 code from Philips. Almost all universal remote controls can be used. The IRCODE variable holds the last received code.
Note: To restart a new receive a null has to be written to the IRCODE variable.

SETPWM <channel>,<divider>,<period>

The SETPWM command initializes and starts a hardware PWM (Pulse Width Modulation). Refer to the hardware manual to check how many PWM channels are supported by the controller. The channel parameter specifies the channel to be initialized. The divider is used to select the appropriate clock source for the internal 16bit timer used to generate the PWM. The period is the length of one cycle in counter ticks.



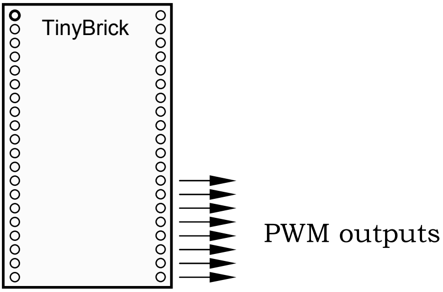
Function: When the internal 16bit timer starts, the output pin is set high. When the PWM time is reached, the output pin is set low. When the period time is reached the timer is restarted.

As the timer is 16bit wide the maximum range for both times is 0..65535 timer ticks. Depending on the source clock almost any frequency is possible: e.g. SETPWM 0,0,1000 generates a frequency of 20kHz and a pulse width of 0..50µs in 1000 steps, while SETPWM 0,4,60000 results in a frequency of 10Hz and a pulse width of 0..100ms in 60000 steps.

Note: TinyBrick16 and TinyDisplay require the same divider for all channels. TinyBrick8 supports 2 deviders – one for channels 0..2 and one for channels 3..5.

PWM*n*

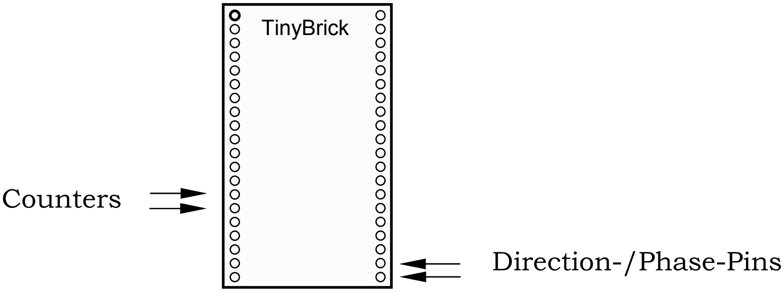
After initialization of a PWM channel the current pulse width may be changed and also read back via the variables PWM0..PWM7. The range of each variable depends on the initialization and can be 0..<period> (refer to command SETPWM).



SETCOUNTER <channel>,<mode>

Two external interrupt pins INT0 and INT1 can be used to work as counters. The SETCOUNTER command configures each channel 0..1 to count either the raising edge, the falling edge or even both edges.

Mode 0	count raising edge only
Mode 1	count falling edge only
Mode 2	count both edges
Mode 3	count both edges up-down (direction pin)
Mode 4	count both edges up-down (quadrature pin)



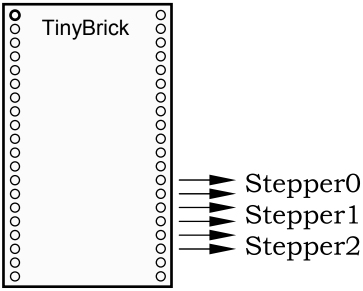
Note: The counter inputs can also be used to restart the controller after executing the END command.

COUNTER*n*

After initialization of one or both counters the variables COUNTER0 and COUNTER1 hold the current counter value. The counters can be read and written at any time. Normally, the counters have to be nulled, e.g. COUNTER0 = 0, but it is also possible to set the counter values to a specific reference value, e.g. COUNTER0 = 1000. Both counters are 16bit wide (0..65535).

SETSTEPPER <motor>,<mode>

TinyBasic can control up to 3 stepper motors. Each motor has a step- and a direction output pin. By now only the start/stop-mode (mode=0) is implemented.



STEPPER*n*
STEPPER*n*.SPEED
STEPPER*n*.STEPS

The STEPPER*n* variable holds the absolute position of the stepper motor and can be read and written at any time. Its 16bit value can be in the range 0..65535 steps.
To move a stepper set STEPPER*n*.SPEED first and then the STEPPER*n*.STEPS. The speed is set in steps/second and can be in the range 50..2000 steps/sec. The steps value can be in the range -65535..65535. If the value is negative the direction pin becomes high. The steps and the speed values can be read at any time. As long as the motor is moving the steps value counts down. If the steps value is set to null while the motor is moving it will be stopped immediately.

TIMER*n*

There are 4 software count down timers TIMER0..TIMER3, each 31bit wide (0..2147483647). The timer tick is one millisecond, this leads to a maximum time span of approx. 24 days.
As soon as a time value is assigned to one of these variables, e.g. TIMER0 = 100, the variable counts down to 0. If the 0 is reached the timer will be stopped. The rest time can always be read, but the safe way to use these timers is to check if the 0 is already reached.

ADC*n*

Analog input pins can be read via variables: ADC0..ADC7. No further pin initialization is required – each time a variable is read the appropriate input pin is configured as analog input. Refer to the pin assignment section which pins of the controller are used for analog input.

SPISHIFT <mode>,<direction>,<variables>

The SPI (Serial Peripheral Interface) is a 3 wire connection between a master and one or more slaves. TinyBasic always works as master and therefore it provides the clock signal. There are four clock modes, due to the combination of clock polarity and clock phase.

Mode 0	positive clock pulse, first latch then shift
Mode 1	positive clock pulse, shift first then latch
Mode 2	negative clock pulse, first latch then shift
Mode 3	negative clock pulse, shift first then latch

The clock mode is selected with the first parameter. The second parameter sets the data direction: 0=output, 1=input and 2=bidirectional data exchange. During data exchange the values of the given variables are replaced by the incomming data.
The number of clock cycles generated results from the variable type. If a byte variable is given only 8 clock pulses will be generated. If a word variable is given 16 clock pulses will be generated and so on. If more than one variable is given then all variables will be clocked out. It is also possible to use array variables then all fields of the array are sent.
Note: Transmission is always performed with MSB (Most Significant Bit) first.

I2CIN <chip>,,<variables>
I2CIN <chip>,<address>,<variables>
I2COUT <chip>,<address>,<variables>

The I²C interface connects multiple ICs via two wires. TinyBasic can be used as an I²C bus master. In order to send the values of one or more variables to one of the connected slaves the I2COUT command can be used. The first parameter is the chip identification – an 8 bit value. I²C slaves only need the upper 7 bits as their address. The lowest bit of the chip identification is used to specify if the slave uses 8 bit or 16 bit adressing mode. The second parameter is the address itself, which depends on the connected slave (EEPROM, RTC...). Finally, a list of variables can be specified. Each variable will be sent via the I²C bus according to its type – bytes with 8 clock pulses, words with 16 clock pulses and so on. Even array variables are supported. This allows to store data set into an external eeprom or serial flash.

To read data from a connected I^{2}C slave the I2CIN command can be used. The chip identification and the address are used in the same way as with the I2COUT command. Some I^{2}C slaves do not need an address therefore the address parameter can be skipped.

SOUND <frequency>
SOUND <frequency>,<duration>

TinyBasic also supports a SOUND statement, to alert the user or even to play a melody. The frequency, given in Hz, is expected as the first parameter. Frequencies in the range 20..20000Hz can be generated. If no duration is given the sound will be very short, approx. 25ms. The duration can be set in the range 1..63, which are steps of 25ms. Therefore, the maximum duration of a sound is 63*25ms approx. 1.5 seconds.
Note: While the SOUND statement is executed the internal soft timer interrupt is halted to avoid distorsions. Therefore, the timer variables TIMER0..TIMER3 are temporarily stopped.

POKE <addr>,<value>

The POKE command allows direct access to all registers of the controller. The registers are mapped into the memory area and can be written as single bytes. Refer to the hardware manual or to the data sheet of the controller to find out the right register address.

STANDBY

To reduce power consumption the easiest way to do so is to use the STANDBY command inside the main loop. STANDBY powers down the controller until the next interrupt occurs. The soft timers are clocked by a 10ms interrupt. That means the maximum time the controller is powerd down is 10ms, but any other interrupt, e.g. the serial receive interrupt or an incoming network message also wake up the controller. Therefore, the main advantage of using STANDBY is, that the program is still fully functioning, while the power consumption is reduced.

END

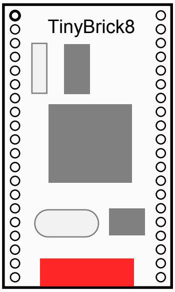
The most dramatically reduction of power consumption is reached by using the END command anywhere in the program. This results in shutting down the controllers clock and therefore reduces the supply current to some µA (microampere). The controller restarts again if either the reset signal is drawn or one of the counter edges is detected.

Note: The counters must be initialized prior to the END command in order to use an external interrupt source (INT0 or INT1) to restart the controller. The program restarts always at the first line.

Note: As this command is shutting down the controller a subsequent program download sequence needs to restart the controller manually.

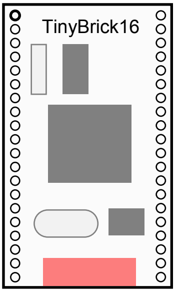
APPENDIX

PIN ASSIGNMENT



RXD(V24)	**+5V**
TXD(V24)	P0.0/ADC7
DTR(V24)	P0.1/ADC6
LCD_CS/P1.0	P0.2/ADC5
LCD_RES/P1.1	P0.3/ADC4
LCD_MODE/P1.2	P0.4/ADC3/COL0
SOUT/P3.7	P0.5/ADC2/COL1
SIN/SCL/P3.3	P0.6/ADC1/COL2
SCLK/P3.5	P0.7/ADC0/COL3
SDA/P3.4	VREF
RXD1(TTL)/P1.5	P3.0
TXD1(TTL)/P1.4	P3.1/SOUND
CLK1/P1.6	P2.0
INT0/CNT0/DCF/P4.5	P2.1/PWM0
INT1/CNT1/IRIN/P1.7	P2.2/PWM1
TX-	P2.3/PWM2
TX+	P2.4/ROW0
RESET	P2.5/PWM3/ROW1
NMI	P2.6/PWM4/ROW2/DIR0
GND	P2.7/PWM5/ROW3/DIR1

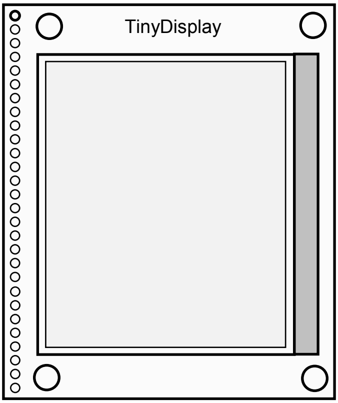
TinyBrick8



RXD(V24)	**+5V**
TXD(V24)	P0.0/ADC0
DTR(V24)	P0.1/ADC1
LCD_CS/P1.5	P0.2/ADC2
LCD_RES/P1.6	P0.3/ADC3
LCD_MODE/P1.7	P0.4/ADC4/COL0
SOUT/P3.2	P0.5/ADC5/COL1
SIN/SCL/P3.1	P0.6/ADC6/COL2
SCLK/P3.0	P0.7/ADC7/COL3
CARD/SDA/P3.3	VREF
RXD1(TTL)/P7.1	P7.4
TXD1(TTL)/P7.0	P7.6/SOUND
CLK1(TTL)/P7.2	P2.0/PWM0
INT0/CNT0/DCF/P8.2	P2.1/PWM1
INT1/CNT1/IRIN/P8.3	P2.2/PWM2
TX-	P2.3/PWM3
TX+	P2.4/PWM4/ROW0
RESET	P2.5/PWM5/ROW1
NMI	P2.6/PWM6/ROW2/DIR0
GND	P2.7/PWM7/ROW3/DIR1

TinyBrick16

+5V
TXD(V24)
RXD(V24)
P3.1/SCL
P3.3/SDA
P8.2/INT0/CNT0/DCF
P8.3/INT1/CNT1/IRIN
P0.0/ADC0
P0.1/ADC1
P0.2/ADC2
P0.3/ADC3
P0.4/ADC4/COL0
P0.5/ADC5/COL1
P0.6/ADC6/COL2
P0.7/ADC7/COL3
P2.0/PWM0
P2.1/PWM1
P2.2/PWM2
P2.3/PWM3
P2.4/PWM4/ROW0
P2.5/PWM5/ROW1
P2.6/PWM6/ROW2/DIR0
P2.7/PWM7/ROW3/DIR1
TX+/TXD1(TTL)/P7.0
TX-/RXD1(TTL)/P7.1
NMI
RESET
GND



TinyDisplay

PIN DESCRIPTION

The following table gives a brief overview about the port pins used by TinyBasic special functions.

Pin	**Pin description**
TXD, RXD, DTR	RS232 interface signals with V24 level, access as device #0
LCD_CS, LCD_RES, LCD_MODE	LCD control signals for chip select, reset and data/ctrl-mode, these pins also control a 7 segment LED display via MAX7219 (LOAD, DATA, CLK resp.)
SIN, SOUT, SCLK	SPI interface, SOUT and SCLK also used for LCD display, also used for SD card
SCL, SDA (CARD)	I²C interface, pull up with 100kΩ, CARD signal is used as SD card chip select
TXD1, RXD1, CLK1	RS232 interface with TTL level, access as device #1 (check jumper settings)
TX+, TX-	RS485 interface (check jumper settings)
INT0/CNT0, INT1/CNT1	Interrupt signals used as counters or wakeup
DIR0, DIR1	Counter direction/phase input pins
DCF	DCF77 radio clock signal
IRIN	Infrared receiver input (RC5 code only)
PWM0..PWM7	Pulse width modulation (PWM) outputs
ADC0..ADC7	Analog input channels
VREF	Analog reference voltage input
COL0..COL3, ROW0..ROW3,	Columns and rows for a 3x4 telephone or a 4x4 numeric keypad (use rectifiers in column lines)
P2.0..P2.6	Stepper motor control pins
P0.0..P0.7	General purpose in/out pins
P2.0..P2.7	General purpose in/out pins
P1.0..P1.7	General purpose in/out pins
P3.0, P3.1, P7.4, P7.6	General purpose in/out pins
NMI	Non maskable interrupt input pin
RESET	Reset input pin