

# Quick Reference Guide for Basic language

# Quick Reference

with  
EXAMPLES

This reference guide is intended to quickly introduce users to BASIC language syntax with the aim to easily start programming microcontrollers along with other applications.

Why BASIC in the first place? The answer is simple: it is legible, easy-to-learn, structured programming language, with sufficient power and flexibility needed for programming microcontrollers. Whether you had any previous programming experience, you will find that writing programs in BASIC is very easy.

Software and Hardware  
solutions for Embedded World



# mikroBasic Quick Reference Guide

## COMMENTS

Any text between an apostrophe and the end of the line constitutes a comment. May span one line only.

### Example:

```
' Put your comment here!  
' It may span one line only.
```

## LITERALS

### Character Literals

Character literal is one character from the extended ASCII character set, enclosed by quotes.

### Example:

```
"A" // this is character A
```

### String Literals

String literal is a sequence of up to 255 characters from the extended ASCII character set, enclosed by quotes.

### Example:

```
"Hello!" ' 12 chars long  
"C" ' 1 char long
```

## KEYWORDS

<b>absolute</b>	<b>float</b>	<b>or</b>
<b>abs</b>	<b>for</b>	<b>org</b>
<b>and</b>	<b>function</b>	<b>print</b>
<b>array</b>	<b>goto</b>	<b>procedure</b>
<b>asm</b>	<b>gosub</b>	<b>program</b>
<b>begin</b>	<b>if</b>	<b>read</b>
<b>boolean</b>	<b>include</b>	<b>select</b>
<b>case</b>	<b>in</b>	<b>sub</b>
<b>char</b>	<b>int</b>	<b>step</b>
<b>chr</b>	<b>integer</b>	<b>string</b>
<b>clear</b>	<b>interrupt</b>	<b>switch</b>
<b>const</b>	<b>is</b>	<b>then</b>
<b>dim</b>	<b>loop</b>	<b>to</b>
<b>div</b>	<b>label</b>	<b>until</b>
<b>do</b>	<b>mod</b>	<b>wend</b>
<b>double</b>	<b>module</b>	<b>while</b>
<b>else</b>	<b>new</b>	<b>with</b>
<b>end</b>	<b>next</b>	<b>xor</b>
<b>exit</b>	<b>not</b>	

### Note:

User can not use keywords for variable or function names. Keywords are reserved only for making basic language statements.



## VARIABLES

### Syntax:

```
dim identifier_list as type
```

### Example:

```
dim i, j, k as byte
```

## CONSTANTS

### Syntax:

```
const constant_name [as type] = value
```

### Example:

```
const MAX as longint = 10000  
const MIN = 1000 ' compiler will assume word type  
const SWITCH = "n" ' compiler will assume char type  
const MSG = "Hello" ' compiler will assume string type
```

## LABELS

### Syntax:

```
label_identifier : statement
```

### Example:

```
loop: Beep ' infinite loop  
goto loop ' that calls the  
' Beep procedure
```

# mikroBasic Quick Reference Guide

## SYMBOLS

**Syntax:**  
**symbol** *alias* = *code*

**Example:**  
**symbol** MAXALLOWED = 216  
*' Symbol as alias for numeric value*  
**symbol** PORT = PORTC  
*' Symbol as alias for SFR*



## FUNCTIONS AND PROCEDURES

### Functions

**Syntax:**  
**sub function** *function\_name*(*parameter\_list*) **as** *return\_type*  
    [ *local declarations* ]  
    *function body*  
**end sub**

**Example:**  
**sub function** add(**dim** a, b **as byte**) **as byte**  
    result = a + b  
**end sub**

We can call it to calculate sum of two numbers:

```
dim c as byte  
c = add(4, 5)
```

Variable c will then be 9.

### Procedures

**Syntax:**  
**sub procedure** *procedure\_name*(*parameter\_list*)  
    [ *local declarations* ]  
    *procedure body*  
**end sub**

**Example:**  
**sub procedure** add(**dim byref** c **as byte**, **dim** a, b **as byte**)  
    c = a + b  
**end sub**



### Note:

When we want the parameter to be changed in function or procedure body then we must declare that parameter with **dim byref** directive instead with **dim** directive. If declared with **dim** directive changes to the parameter will take effect only in that function or procedure.

We can call it to calculate sum of two numbers (last two parameters) and place result in first parameter:

```
dim c as byte  
add(c, 4, 5)
```

Variable c will then be 9.



## SIMPLE TYPES

Type	Size	Range
byte	8-bit	0 .. 255
char	8-bit	0 .. 255
word	16-bit	0 .. 65535
short	8-bit	- 128 .. 127
integer	16-bit	-32768 .. 32767
longint	32-bit	-2147483648 .. 2147483647
float	32-bit	$\pm 1.17549435082 * 10^{-38}$ ... $\pm 6.80564774407 * 10^{38}$

## ARRAYS

**Syntax:**  
`type[array_length]`

**Example:**  
`dim weekdays as byte[7]`  
`dim samples as word[10]`  
*' now we can access elements of array variables, for example:*  
`samples[0] = 1`  
`if weekdays[1] = 1 then`    *' if it is Tuesday (day with index 1)*  
     `samples[0] = 20`        *' then order 20 samples with index 0*  
     `else samples[3] = 10`    *' else order 10 samples with index 3*  
`end if`

## CONSTANT ARRAYS

**Example:**  
*' Declare a constant array which holds no. of days*  
*' in each month:*  
`const MONTHS as byte[12] = (31,28,31,30,31,30,31,31,30,31,30,31)`

## STRINGS

**Syntax:**  
`string[string_length]`

**Example:**  
`dim msg1 as string[20]`  
`dim msg2 as string[19]`  
**begin**

`msg1 = "First message"`  
`msg2 = "Second message"`  
`msg1 = msg2`

This is ok, but vice versa would be illegal (because length of string msg1 is greater then length of string msg2).

## POINTERS

To declare a pointer data type, add a carat prefix (^) before type. For example, if you are creating a pointer to an integer, you would write:

`^integer`

A pointer can be assigned to another pointer. However, note that only the address, not the value, is copied.

**Example:**  
`dim p as ^word`  
`...`  
`p^ = 5`

*This will assign the pointed memory location value 5.*

**@ Operator**

The @ operator returns the address of a variable or routine; that is, @ constructs a pointer to its operand. The following rules apply to @:

- If X is a variable, @X returns the address of X.
- If F is a routine (a function or procedure), @F returns F's entry point

## STRUCTURES

**Syntax:**  
`structure structname`  
     `dim member1 as type1`  
     `...`  
     `dim membern as typen`  
`end structure`

**Example:**  
`structure Dot`  
     `dim x as float`  
     `dim y as float`  
`end structure`

# mikroBasic Quick Reference Guide



Memory is allocated when you instantiate the record, like this:

```
dim m as Dot
dim n as Dot
```



## Accessing Fields

**Example:**  
`m.x = 3.6`  
`m.y = 5`



## Note:

You can commit assignments between complex variables, if they are of the same type:

```
n = m
```

This will copy values of all fields.



## OPERATORS

There are four types of operators in mikroBasic:

- Arithmetic Operators
- Bitwise Operators
- Boolean Operators
- Relational Operators

### Operators Precedence and Associativity

Precedence	Operands	Operators	Associativity
4	1	@ not + -	right-to-left
3	2	div mod * / and << >>	left-to-right
2	2	+ - or xor	left-to-right
1	2	= <> < > <= >=	left-to-right

### Arithmetic Operators

Operator	Operation	Precedence
+	addition	2
-	subtraction	2
*	multiplication	3
/	division	3
div	division, rounds down to nearest integer (cannot be used with floating points)	3
mod	returns the remainder of integer division (cannot be used with floating points)	3

### Relational Operators

Operator	Operation	Precedence
=	equal	1
<>	not equal	1
>	greater than	1
<	less than	1
>=	greater than or equal	1
<=	less than or equal	1



## Note:

Use relational operators to test equality or inequality of expressions. All relational operators return TRUE or FALSE.

## Bitwise Operators

Operator	Operation	Precedence
and	bitwise AND; returns 1 if both bits are 1, otherwise returns 0	3
or	bitwise (inclusive) OR; returns 1 if either or both bits are 1, otherwise returns 0	2
xor	bitwise exclusive OR (XOR); returns 1 if the bits are complementary, otherwise 0	2
not	bitwise complement (unary); inverts each bit	4
<<	bitwise shift left; moves the bits to the left, see below	3
>>	bitwise shift right; moves the bits to the right, see below	3

### Examples:

```
operand1 :      %0001 0010
operand2 :      %0101 0110
-----
operator and :  %0001 0010
operator or  :  %0101 0110
operator xor  :  %0100 0100
```

### Examples:

```
operand :      %0101 0110
-----
operator not  :  %1010 1001
operator <<  :  %1010 1100
operator >>  :  %0010 1011
```



### Note:

With shift left (<<), left most bits are discarded, and "new" bits on the right are assigned zeroes. With shift right (>>), right most bits are discarded, and the "freed" bits on the left are assigned zeroes (in case of unsigned operand) or the value of the sign bit (in case of signed operand).

## Boolean Operators

Operator	Operation
and	logical AND
or	logical OR
xor	logical exclusive OR
not	logical negation

These operators conform to standard Boolean logic. If used in conditional expressions they are compared with TRUE or FALSE.

### Example:

```
if (%1001 and %0111) = FALSE then LED1 = 1 else LED2 = 1 end if
```

Because expression (%1001 and %0111) gives %0001, when compared with FALSE (all zeros) it gives FALSE because they are not equal. It means that else statement will be executed and LED2 will be turned on. If it was written like this:

```
if (%1001 and %0111) then LED1 = 1 else LED2 = 1 end if
```

than expression (%1001 and %0111) is compared with TRUE (all ones) by default.

# mikroBasic Quick Reference Guide

## STATEMENTS

### asm Statement

**Syntax:**

```
asm
    block of assembly instructions
end asm
```

### Assignment Statements

**Syntax:**

```
variable = expression
```

**Example:**

```
counter = 1
```

## CONDITIONAL STATEMENTS

### If Statement

**Syntax:**

```
if expression then
    statements
[else
    other statements]
end if
```

**Example:**

```
if movex = 1 then
    x = x + 20
else
    y = y - 10
end if
```

**Note:**

The else keyword with an alternate statement is optional.



### Select Case Statement

**Syntax:**

```
select case selector
    case value_1
        statements_1
    ...
    case value_n
        statements_n
[case else
    default_statements]
end select
```

**Example:**

```
select case input
    case 1
        LED1 = 1
    case 2
        LED2 = 1
    case 3
        LED3 = 1
    case else
        LED7 = 1
end select
```

**Note:**

This code will turn on LED depending of input value. If the value is different then ones mentioned in value list in case statement then else statement is executed by default.



## ITERATION STATEMENTS (LOOPS)

### For Statement

**Syntax:**

```
for counter = initial_value to final_value [step step_value]
    statements
next counter
```

**Example:**

```
s = 0
for i = 0 to 4
    s = s + 2
next i
```

This code will add number 2 to variable s 5 times. At the end s will be 10.

**Note:**

step directive is optional. It defines incrementing (or decrementing if negative) value of counter after each iteration. Default step value is 1.



### While Statement

**Syntax:**

```
while expression
    statements
wend
```

**Example:**

```
s = 0
i = 0
while i < 6
    s = s + 2
    i = i + 1
wend
```

This code will add number 2 to variable s 6 times. At the end s will be 12.

## JUMP STATEMENTS

### Do Statement

#### Syntax:

```
do
    statements
loop until expression
```

#### Example:

```
s = 0
i = 0
do
    s = s + 2
    i = i + 1
loop until i = 7
```

This code will add number 2 to variable s 7 times. At the end s will be 14.

### Break Statement

Use the break statement within loops to pass control to the first statement following the innermost loop (for, while, and do).

#### Example:

```
i = 0           ' initiate value of counter i
s = 1           ' and variable s
while true     ' infinite loop
    if i = 4 then break
    end if
    s = s * 2
    i = i + 1
wend
```

This code will multiply variable s with number 2 (until counter i becomes equal 4 and break statement executes). At the end s will be 16.

### Continue Statement

You can use the continue statement within loops to skip the rest of the statements and jump to the first statement in loop.

#### Example:

```
i = 0           ' initiate value of counter i
s = 1           ' and variable s
while true     ' infinite loop
    s = s * 2
    i = i + 1
    if i <> 4 then continue
    end if
    break
wend
```

This code will multiply variable s with number 2 (continue statement executes until counter i is not equal 4). At the end s will be 16.

### Goto Statement

#### Syntax:

```
goto label_name
```

#### Example:

```
loop: Beep      ' infinite loop
goto loop     ' that calls the
                ' Beep procedure
```

### Goto Statement

#### Syntax:

```
gosub label_name
```

```
...
label_name:
...
return
```

### Exit Statement

The exit statement allows you to break out of a routine (function or procedure). It passes the control to the first statement following the routine call.

#### Example:

```
sub procedure Proc1()
dim error as byte
... ' we're doing something here
if error = TRUE then exit
end if
... ' some code, which won't be
end sub ' executed if error is true
```