QUICK REFERENCE GUIDE FOR mikroBasic

MikroElektronika
SOFTWARE AND HARDWARE SOLUTIONS FOR EMBEDDED WORLD ...making it simple

Lexical Elements Overview

The mikroBasic quick reference guide provides formal definitions of lexical elements the mikroBasic programming language consists of. These elements are word-like units recognized by such programming language. Every program written in mikroBasic consists of a sequence of ASCII characters such as letters, digits and special signs. Non-printing signs such as newline characters, tab etc. are also referred to as special signs. A set of basic elements in mikroBasic is well-organized and finite. Programs are written in the *mikroBasic Code Editor* window. During the process of compiling, each file code is parsed into tokens and whitespaces.

Whitespace

Spaces (blanks), horizontal and vertical tabs and newline characters are together called whitespaces. Whitespaces are used as separators to indicate where tokens start and end. For example, the two following sequences

```
dim tmp as byte
dim j as word
```

and

```
dim  tmp  as  byte
dim    j  as  word
```

are lexically equivalent and parse identically giving the eight tokens each:

```
dim
tmp
as
byte
dim
j
as
word
```

Whitespace in Strings

Whitespace may occur within string literals. In this case it is not used as a parser, but is construed as a common character, i.e. represents part of the string alone. For example, the following string

```
some_string = "mikro foo"
```

parses into four tokens, including the string literal as a single token:

```
some_string
=
"mikro foo"
newline character
```

Tokens

A token is the smallest element of the Basic programming language which is recognized by the compiler. The parser analyzes the program code from left to right and creates the longest possible tokens from the sequence of input characters.

Keywords

Keywords or reserved words are tokens with fixed meaning which cannot be used as identifiers. In addition to the standard mikroBasic keywords, there is a set of predefined identifiers (of constants and variables) referred to as reserved words. They describe specific microcontroller and cannot be redefined. Here is a list of the mikroBasic keywords in alphabetical order:

absolute	cstr	form	let	private	stop
and	curdir	format	line	procedure	str
appactivate	currency	forward	loc	program	strcomp
array	cvar	freefile	lock	property	strconv
as	cverr	function	lof	pset	string
asc	data	fv	long	public	structure
asm	date	get	longint	put	sub
at	dateadd	getattr	longword	pv	switch
atn	datediff	getobject	loop	qbcolor	syd
attribute	datepart	gosub	lset	raise	symbol
base	dateserial	goto	me	randomize	system
bdata	datevalue	hex	mid	rate	tab
beep	ddb	hour	minute	redim	time
bit	deftype	idata	mirr	register	timer
boolean	dim	if	mkdir	rem	timeserial
byref	dir	iif	mod	resume	timevalue
byte	div	ilevel	module	return	to
call	do	imp	month	rgb	typedef
case	doevents	implements	msgbox	right	typename
cbool	double	include	name	rmdir	ubound
cbyte	each	input	new	rnd	ucase
ccur	eio	instr	next	rset	unlock
cdate	empty	int	not	rx	until
cdate	end	integer	not	sbit	val
cdbl	end with	io	nothing	second	variant
char	environ	ipmt	now	seek	vartype
chdir	eof	irr	nper	select	version
chdrive	eqv	is	npv	sendkeys	volatile
chr	erase	isarray	object	set	weekday
cint	err	isdate	oct	setattr	wend
circle	error	isempty	on	sfr	while
class	exit	iserror	open	sgn	width
clear	explicit	ismissing	option	shell	with
clng	explicit	isnull	option	short	word
close	fileattr	isnumeric	option	single	write
code	fileattr	isobject	or	sln	xdata
command	filecopy	kill	org	small	xor
compact	filedatetime	large	orgall	space	
compare	filelen	lbound	pdata	spc	
const	fix	lcase	pmt	sqr	
createobject	float	left	ppmt	static	
csng	for	len	print	step	

Comments

Comments are part of the program used for clarifying program operation or for providing more information about it. They are exclusively intended for the programmer's use and are removed from the program before parsing. There are only single-line comments in mikroBasic starting with an apostrophe '. Here is an example:

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

Besides, blocks of assembly instructions may introduce single-line comments by placing ';' before the comment:

```
asm
  some_asm ; This assembly instruction …
end asm
```

Identifiers

Identifiers are arbitrary names used for designating the basic language objects (labels, types, constants, variables, procedures and functions). Identifiers may contain all the letters of alphabet (both upper case and lower case), the underscore character '_' and digits (0 to 9). The first character of an identifier must be a letter or an underscore. mikroBasic is not case sensitive, so that Sum, sum, and suM are recognized as equivalent identifiers. Although identifier names are arbitrary (within the rules stated), some errors may occur if the same name is used for more than one identifier within the same scope. Here are some valid identifiers:

```
temperature_V1
Pressure
no_hit
dat2string
SUM3
_vtext
```

Here are some invalid identifiers:

```
7temp         ' NO -- cannot begin with a numeral
%higher       ' NO -- cannot contain special characters
xor           ' NO -- cannot match reserved word
j23.07.04     ' NO -- cannot contain special characters (dot)
```

Literals

Literals are tokens representing fixed numerical or character values. The compiler determines data type of a literal on the basis of its format (the way it is represented in the code) and its numerical value.

Integer Literals

Integral literals can be written in decimal, hexadecimal or binary notation.

- In decimal notation, integer literals are represented as a sequence of digits (without commas, spaces or dots), with optional prefix + or - (operator). Integer literals without prefix are considered positive. Accordingly, the number 6258 is equivalent to the number +6258.
- Integer literals with the dollar sign ($) or 0x prefix are considered hexadecimal numbers. For example, $8F or 0x8F.
- Integer literals with the percent sign prefix (%) are considered binary numbers. For example, %01010101.

Here are some examples of integer literals:

```
11       ' decimal literal
$11      ' hex literal equal to decimal 17
0x11     ' hex literal equal to decimal 17
%11      ' binary literal equal to decimal 3
```

The allowed range for constant values is determined by the **longint** type for signed constants or by the **longword** type for unsigned constants.

Floating Point Literals

A floating point literal consists of:

- Decimal integer;
- Decimal point;
- Decimal fraction; and
- e or E and a signed integer exponent (optional).

Here are some examples of floating point literals:

```
0.         ' = 0.0
-1.23      ' = -1.23
23.45e6    ' = 23.45 * 10^6
2e-5       ' = 2.0 * 10^-5
3E+10      ' = 3.0 * 10^10
.09E34     ' = 0.09 * 10^34
```

Character Literals

Character literals are ASCII characters, enclosed in single quotation marks. Character literals may be assigned to variables of integral and string type, array of characters and array of bytes. A variable of integral type will be assigned the ASCII value of the corresponding character literal.

String Literals

String literal represents a sequence of ASCII characters written in one line and enclosed in single quotation marks. As mentioned before, string literals may contain whitespaces. The parser does not “go into” string literals, but treats them as single tokens. The length of a string literal depends on the number of characters it consists of. There is a null character (ASCII zero) at the end of each string literal. It is not included in the string’s total length. A string literal with nothing in between the quotation single marks (null string) is stored as one single null character. String literals can be assigned to string variables, array of char or array of byte. Here are several string literals:

```
"Hello world!"                    ' message, 12 chars long
"Temperature is stable"           ' message, 21 chars long
" "                               ' two spaces, 2 chars long
"C"                               ' letter, 1 char long
""                                ' null string, 0 chars long
```

Punctuators

mikroBasic uses the following punctuators (also known as separators):

- [ ] - square brackets;
- ( ) - parentheses;
- , - comma;
- ; - semicolon (with the **asm** statements only);
- : - colon
- . - dot

Square brackets

Brackets [ ] are used for indicating single and multidimensional array’s indices:

```
dim alphabet as byte [ 30]
' ...
alphabet [ 2] = "c"';
```

Parentheses

Parentheses ( ) are used for grouping expressions, isolating conditional expressions and for indicating function calls and function declarations:

```
d = c * (a + b)                   ' Group expressions
if (d = z) then ...               ' Conditional expressions
func()                            ' Function call, no parameters
sub function func2(dim n as word) ' Function declaration with parameters
```

Comma

Commas ‘,’ are used for the purpose of separating parameters in function calls, identifiers in declarations and elements of initializer:

```
Lcd_Out(1, 1, txt)
dim i, j, k as word
const MONTHS as byte [ 12] = (31,28,31,30,31,30,31,31,30,31,30,31)
```

Semicolon
A semicolon is used for denoting points in assembly blocks at which comments start.

Colon
A colon (:) is used for indicating a label in the program. For example:

```
start:  nop
goto start
```

Dot
A dot (.) is used for indicating access to a structure member. It can also be used for accessing individual bits of registers. For example:

```
person.surname = "Smith"
```

Program Organization

Similar to other programming languages, mikroBasic provides a set of strictly defined rules to be observed when writing programs. In other words, all the programs written in mikroBasic have a clear and organized structure. Some of the program examples are given in text below. Every project written in mikroBasic consists of a project file and one or more modules (files with the **.mbas** extension). The project file provides information on the project itself, whereas modules contains program code.

Main Module
Every project in mikroBasic requires a single main module. It is identified by the **program** keyword placed at the beginning of the module, thus instructing the compiler from where to start the process of compiling. After an empty project is successfully created in *Project Wizard*, the main module will be automatically displayed in the *Code Editor* window. It contains the basic structure of the program written in mikroBasic. Nothing may precede the **program** keyword, except comments. The **include** clause may optionally be placed after the program name. All global identifiers of constants, variables, labels and routines are declared before the **main** keyword.

Structure of the Main Module

Basically, the main module can be divided in two sections: declarations and program body. Declarations must be properly organized and placed in the code. Otherwise, the compiler may not be able to interpret the program correctly. When writing a code, it is advisable to follow the model presented below.

The main module is to look as follows:

```
program <program_name>
include <include_other_modules>

'******************************************************
'* Global declarations:
'******************************************************

'symbol declarations
symbol ...

'constant declarations
const ...

'variable declarations
dim ...

'procedure declarations
sub procedure procedure_name (...)
  <local_declarations>
    ...
  end sub

'function declarations
sub function function_name (...)
  <local_declarations>
    ...
  end sub

'******************************************************
'* Program body:
'******************************************************

main:
  'write your code here
end.
```

Other modules

Other modules allow you to:

- break large programs into encapsulated parts that can be edited separately;
- create libraries that can be used in different projects; and
- distribute libraries to other developers without disclosing the source code.

Every module is stored in its own file and compiled separately. Compiled modules are linked together for the purpose of creating an executable code. To build a project correctly, the compiler needs all the modules either as program code files or object files (files created by compiling modules). All modules start with the **module** keyword. Apart from the comments, nothing may precede the **module** keyword. The **include** clause may follow the module name.

Structure of Other Modules

Every module consists of three sections. These are **include**, **interface** and **implementation** sections. Only the **implementation** section is obligatory. Follow the model presented below:

```
module   <module_name>
include <include_other_modules>
'*********************************************************
'* Interface (globals):
'*********************************************************

' symbol declarations
symbol ...

' constant declarations
const ...

' variable declarations
dim ...

' procedure prototypes
sub procedure procedure_name(...)

' function prototypes
sub function function_name(...)

'*********************************************************
'* Implementation:
'*********************************************************

implements
' constant declarations
const ...

' variable declarations
dim ...

' procedure declarations
sub procedure procedure_name (...)
  <local_declarations>
    ...
  end sub

' function declarations
sub function function_name (...)
  <local_declarations>
    ...
  end sub

end.
```

Include Clause

mikroBasic includes modules by means of the **include** clause. It consists of the **include** reserved word and one module name enclosed in quotation marks. Module name does not include extension. Every module may include more than one **include** clause which must be placed immediately after the module name. Here is an example:

```
program MyProgram

include "utils"
include "strings"
include "MyUnit"
...
```

After reaching the module name, the compiler checks if the project contains files with the given name and **.mcl** and **.mbas** extensions in the order and places specified in the *Search Paths* option.

- If both **.mbas** and **.mcl** files are found, the compiler will check their dates and include a file of later date in the project. If the **.mbas** file is of later date, the compiler will recompile it and generate new **.mcl** file by copying the former one;
- If only **.mbas** file is found, the compiler will recompile it and generate the **.mcl** file;
- If only **.mcl** file is found, the compiler will include it as found;
- If no file is found, the compiler will report an error.

Interface Section

A part of the module above the **implements** keyword is referred to as **interface** section. It contains global declarations of constants, variables and symbols for the project. Routines cannot be defined in this section. Instead, the section contains declarations of routines, defined in the **implementation** section, which are to be visible beyond the module. Routine declarations must completely match definitions thereof.

Implementation Section

The **Implementation** section contains private routine declarations and definitions and allows code encapsulation. Everything declared below the **implements** keyword is for private use only, i.e. has the scope limited to that module and thus can be used in any block or routine defined within that module. Any identifier declared within this section of the module cannot be used beyond that module, only in routines defined after declaration of the given identifier.

Scope and Visibility

Scope

The scope of an identifier represents the part of the program in which the identifier can be used. There are different categories of scope depending on how and where identifiers are declared:

Place of declaration	Scope
Identifier is declared in sections for declaring main module, beyond function or procedure.	Scope extends from the point where identifier is declared to the end of the current block, including all routines within the scope.
Identifier is declared within function or procedure.	Scope extends from the point where identifier is declared to the end of that routine. These identifiers are referred to as local identifiers.
Identifier is declared in the interface section of the module.	Scope extends from the point where identifier is declared to the end of the module, as well as to all other modules or programs using that module. The only exceptions are symbols the scope of which is limited to the module in which they are declared.
Identifier is declared in the implementation section of the module, but not within function and procedure.	Scope extends from the point where identifier is declared to the end of the current module. The identifier is available to any function or procedure defined below its declaration.

Visibility

Similar to scope, the visibility of an identifier represents the part of the program in which the identifier can be used. Scope and visibility usually coincide, though there are some situations in which an object referred to by an identifier becomes temporarily hidden by its duplicate (an identifier with the same name, but different scope). In this case, the object still exists, but cannot be accessed by its original identifier until the scope of the duplicate identifier ends. The visibility cannot exceed the scope, but the scope can exceed the visibility.

Types

mikroBasic is strictly typed language, which means that each variable and constant must have its type defined before the process of compiling starts. Checking type may prevent objects from being illegally assigned or accessed.
mikroBasic supports standard (predefined) data types such as signed and unsigned integers of various sizes, arrays, strings, pointers etc. Besides, the user can define new data types using the **typedef** directive. For example:

```
Typedef MyType1 as byte
Typedef MyType2 as integer
Typedef MyType3 as ^word
Typedef MyType4 as ^MyType1

dim  mynumber as MyType2
```

Simple Types

Simple types represent types that cannot be broken down into more basic elements. Here is an overview of simple types in mikroBasic:

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
longword	32–bit	0-4294967295
float	32-bit	$\pm 1.17549435082 * 10^{-38} .. \pm 6.80564774407 * 10^{-38}$

* char type can be treated as byte type in every aspect

Arrays

An array represents a finite and arranged set of variables of the same type called elements. Type of elements is called the base type. The value of an element can be accessed by its index which is unique for each element so that different elements may contain the same value.

Array Declaration

Arrays are declared in the following way:

```
element_type [ array_length]
```

Each element of an array is numbered from 0 to **array_length -1**. The **element_type** specifier represents the type of array elements (the base type). Each element of an array can be accessed by specifying array name followed by the element index enclosed in square brackets. Here are a few examples:

```
dim weekdays as byte [ 7]
dim samples as word [ 50]

main:
   ' Array elements may be accessed in the following way:
   samples [ 0]  = 1
   if samples [ 37]  = 0 then
     ...
```

Constant Arrays

Constant array is initialized by assigning it a comma-delimited sequence of values enclosed in parentheses. For example:

```
'Declare a constant array which holds number of days in each month:
const MONTHS as byte [ 12]  = (31,28,31,30,31,30,31,31,30,31,30,31)
'Declare a 2-dimensional constant array:
const NUMBER s byte [ 4][ 4]  = ((0, 1, 2, 3), (5, 6, 7, 8), (9, 10, 11, 12), (13, 14, 15, 16))
```

The number of assigned values must not exceed the specified array length, but can be less. In this case, the trailing “excess” elements will be assigned zeroes.

Multi-dimensional Arrays

An array is one-dimensional if it is of scalar type. One-dimensional arrays are sometimes referred to as vectors. Multidimensional arrays are created by declaring arrays of array type. Here is an example of a 2-dimensional array:

```
dim m as byte [ 50][ 20]     '2-dimensional array of 50x20 elements
```

The **m** variable represents an array of 50 elements which in turn represent arrays of 20 elements of **byte** type each. This is how a matrix of 50x20 elements is created. The first element is m[0][0], whereas the last one is m[49][19]. If an array is a function parameter then it has to be passed by reference as in the following example:

```
sub procedure example(dim byref m as byte [ 50][ 20] )
...
inc(m[ 1][ 1] )
end sub

var
dim m as byte [ 50][ 20]      ' 2-dimensional array of size 50x20
dim n as byte [ 4][ 2][ 7]    ' 3-dimensional array of size 4x2x7
main:
...
func(m)
end.
```

Strings

A string represents a sequence of characters and is equivalent to an array of **char**. It is declared in the following way:

```
string [ string_length]
```

The **string_name** specifier represents a string name and must be valid identifier. The **string_length** specifier represents the number of characters the string consists of. At the end of each string, there is a final null character (ASCII code 0) which is not included in string's total length. A null string ('') represents an array containing a single null character. Strings can be assigned to variables of string type, array of char and array of byte. For example:

```
dim message1 as string [ 20]
dim message2 as string [ 19]
main:
  msg1 = "This is the first message"
  msg2 = "This is the second message"
  msg1 = msg2
```

A string can also be accessed element–by–element. For example:

```
dim s as string [5]
...
s = "mik"
's[0] is char literal "m"
's[1] is char literal "i"
's[2] is char literal "k"
's[3] is zero
's[4] is undefined
's[5] is undefined
```

String Splicing

mikroBasic allows you to splice strings by means of the plus operator '+'. This kind of concatenation may apply to string variables, string literals, character variables and character literals. Non-printing characters can be represented by means of the **Chr** operator and a number representing the ASCII code thereof (e.g. Chr(13) for CR). For example:

```
dim message as string [100]
dim res_txt as string [5]
dim res, channel as word
main:
res = Adc_Read(channel)        ' ADC result
WordToStr(res, res_txt)        ' Create string out of numeric result
                               ' Prepare message for output
message  = "Result is" +       ' Text "Result is"
       Chr(13)        +       ' Append CR sequence
       Chr(10)        +       ' Append LF sequence
       res_txt        +       ' ADC result
       "."                    ' Append a dot
```

mikroBasic includes the String Library which facilitates string related operations.

Pointers

A pointer is a variable which holds memory address of an object. While variable directly accesses some memory address, the pointer can be thought of as a reference to that address. To declare a pointer, it is necessary to add a carat prefix (^) before its type. For example, to declare a pointer to an object of **integer** type, it is necessary to write:

```
^integer
```

To access data stored at the memory location pointed to by a pointer, it is necessary to add a carat suffix '**^**' to the pointer name. For example, let's declare the **p** variable which points to an object of **word** type, and then assign value 5 to the object:

```
dim p as ^word
...
p^ = 5
```

A pointer can be assigned to another pointer. It makes both pointers point to the same memory location. Modifying the object pointed to by one pointer causes another object pointed to by another pointer to be automatically changed since they share the same memory location.

@ Operator

The **@** operator returns the address of a variable or routine, i.e. directs a pointer to its operand. The following rules apply to this operator:

- If X is a variable, @X returns the address of X.

If variable X is of **array** type, the **@** operator will return the pointer to its first basic element, except when the left side of the statement in which X is used is array pointer. In this case, the **@** operator will return the pointer to the array, not to its first basic element:

```
typedef array_type as byte[ 10]

dim w as word
      ptr_b as ^byte
      ptr_arr as array_type
      arr as byte[ 10]

main:
  ptr_b = @arr            ' @ operator will return ^byte
  w_ = @arr               ' @ operator will return ^byte
  ptr_arr = @arr          ' @ operator will return ^array[ 10]  of byte
end.
```

- If F is a routine (function or procedure), @F returns the pointer to F.

Function Pointers

mikroBasic allows the use of function pointers. This example illustrates how to define and use function pointers. We will first define procedural type and function pointer, then call the function by means of the pointer.

```
' Procedural type definition
 type TMyFunctionType = function (dim param1, param2 as byte, dim param3 as word) as word

dim MyPtr as ^TMyFunctionType   ' Pointer to previously defined procedural type
dim sample as word

' Function definition
' Function prototype should match type definition
sub function Func1(dim p1, p2 as byte, dim p3 as word) as word '
    result = p1 and p2 or p3                  ' return value
 end sub

sub function Func2(dim abc, def as byte, dim ghi as word) as word
  result =  abc * def + ghi                   ' return value
end sub

 sub function Func3(dim first, yellow as byte, dim monday as word) as word '
  result = monday - yellow - first            ' return value
end sub

' main program:
main:
' MyPtr now points to Func1
  MyPtr  = @Func1

' Call function Func1 via pointer
  Sample = MyPtr^(1, 2, 3)

' MyPtr now points to Func2
  MyPtr  = @Func2

' Call function Func2 via pointer
  Sample = MyPtr^(1, 2, 3)

' MyPtr now points to Func3
  MyPtr = @Func3

' Call function Func3 via pointer
  Sample = MyPtr^(1, 2, 3)
 end.
```

Structure

A structure represents a heterogeneous set of elements. Each element is called a member. Structure declaration specifies the name and type of each member. For example:

```
structure structname
    dim member1 as type1
    ...
    dim membern as typen
end structure
```

The **structname** specifier represents a structure name which must be valid identifier, specifiers **member1..membern** represent lists of comma-delimited structure member identifiers, whereas specifiers **type1.. typen** represent types of appropriate structure members. The scope of a member identifier is limited to the structure in which it occurs, so that it is not necessary to take care of naming conflicts between member identifiers and other variables. Note that in mikroBasic the structure can be declared as a new type only.
For example, the following declaration creates a structure called **Dot**:

```
structure Dot
   dim x as float
   dim y as float
end structure
```

Each **TDot** contains two members: **x** and **y** coordinates. Memory is not allocated until an object of **structure** type is defined as in the following example:

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

Such declaration creates two instances of **Dot**, called **m** and **n**. A structure member can be previously defined structure. For example:

```
' Structure defining a circle:
structure Circle
   dim radius as float
   dim center as Dot
end structure
```

Accessing Structure Members

Structure members may be accessed by means of the dot (.) operator. If we declare variables **circle1** and **circle2** of the previously defined type **Circle**,

```
dim circle1, circle2 as Circle
```

their individual members can be accessed in the following way:

```
circle1.radius = 3.7
circle1.center.x = 0
circle1.center.y = 0
```

Type Conversions

Conversion of an object of one type is the process of changing its type into another type. mikroBasic supports both implicit and explicit conversions of basic types.

Implicit Conversion

The compiler automatically performs implicit conversion in the following situations:

- if a statement requires an expression of particular type, but expression of different type is used;
- if an operator requires an operand of particular type, but operand of different type is used;
- if a function requires a formal parameter of particular type, but is assigned an object of different type; and
- if a function result does not match the declared function return type.

Promotion

When operands are of different types, implicit conversion promotes a less complex to a more complex type as follows:

bit → byte/char
byte/char → word
short → integer
short → longint
integer → longint
integral → float
word → longword

Higher bytes of an extended unsigned operand are filled with zeroes. Higher bytes of an extended signed operand are filled with a bit sign. If the number is negative, higher bytes are filled with ones, otherwise with zeroes. For example:

```
dim a as byte
dim b as word
...
a = $FF
b = a   ' a is promoted to word, b to $00FF
```

Clipping

In assignment statements and statements requiring an expression of particular type, the correct value will be stored in destination only if the result of expression doesn't exceed the destination range. Otherwise, excess data, i.e. higher bytes will simply be clipped (lost).

```
dim i as byte
dim j as word
...
j = $FF0F
i = j    ' i becomes $0F, higher byte $FF is lost
```

Explicit Conversion

Explicit conversion can be executed upon any expression by specifying desired type (**byte**, **word**, **short**, **integer**, **longint, longword** or **float**) before the expression to be converted. The expression must be enclosed in parentheses. A special case represents conversion between signed and unsigned types. Such explicit conversion does not affect binary representation of data. For example:

```
dim a as byte
dim b as short
...
b = -1
a = byte(b)   ' a is 255, not 1
' Data doesn't change it's binary representation 11111111
' it's just interpreted differently by the compiler
```

Explicit conversion cannot be performed upon the operand to the left of the assignment operator:

```
word(b) = a   ' Compiler will report an error
```

Here is an example of conversion:

```
dim a, b, c as byte
dim cc as word
...
a = 241
b = 128

c  = a + b          ' equals 113
c  = word(a + b)    ' equals 369
cc = a + b          ' equals 369
```

Note: Conversion of floating point data into integral data (in assignment statements or via explicit typecast) produces correct results only if the float value does not exceed the scope of destination integral type.

Variables

A variable is an object the value of which can be changed during the runtime. Every variable is declared under unique name which must be a valid identifier. Variables can be declared in the file and routine declaration sections. Each variable needs to be declared before it is used in the program. Global variables (visible in all files) are declared in the the main module declaration sections. It is necessary to specify data type for each variable. The basic syntax of variable declaration is:

```
dim identifier_list as type
```

The **identifier_list** specifier represents a list of comma-delimited variable identifiers, whereas the **type** specifier represents their type. mikroBasic allows a shortened version of the syntax comprising of the **dim** keyword followed by multiple variable declarations. For example:

```
dim i, j, k as byte
    counter, temp as word
    samples as longint [ 100]
```

Constants

A constant is an object the value of which cannot be changed during the runtime. RAM memory is not used for their storage. Constants are declared in the file and routine declaration sections as follows:

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

Every constant is declared under unique name (specifier **constant_name**) which must be a valid identifier. Constant names are usually written in uppercase. When declaring a constant, it is necessary to specify its value matching the given type. Specifying the type is optional. In the absence of type, the compiler assumes a simple type with the smallest scope that can accommodate the constant value. mikroBasic allows shortened version of the syntax comprising of the **const** keyword followed by multiple constant declarations. For example:

```
const MAX as longint = 10000
  MIN = 1000        ' compiler will assume word type
  SWITCH = "n"      ' compiler will assume char type
  MSG = "Hello"     ' compiler will assume string type
  MONTHS as byte [ 12]  = (31,28,31,30,31,30,31,31,30,31,30,31)
```

Labels

Labels serve as targets for ***goto*** and ***gosub*** statements. Mark a desired statement with a label like this:

```
label_identifier : statement
```

Label declaration is optional in mikroBasic. Label name must be valid identifier. A signed label and **goto/gosub** statements referring to that label must belong to the same block. For this reason, no jump to or from the procedure or function can be executed. A label can be declared only once within the block. Here is an example of an endless loop calling the **Beep** procedure:

```
loop: Beep
goto loop
```

Symbols

Symbols in mikroBasic enable simple macros without parameters to be created. Any code line may be substituted by one identifier. When properly used, symbols may contribute to code legibility. Symbols may be declared at the beginning of the module, below the module name and optional ***include*** directive.

```
symbol alias  = code
```

Symbols are not stored in RAM memory. Instead, each symbol is replaced by the code assigned to it at declaring symbol.

```
symbol MAXALLOWED = 216           'Symbol for numerical value
symbol PORT = PORTC               'Symbol for SFR
symbol MYDELAY = Delay_ms(1000)   'Symbol for procedure call
dim cnt as byte                   'Some variable
'...
main:
if cnt > MAXALLOWED then
  cnt = 0
  PORT.1 = 0
  MYDELAY
end if
```

Functions and Procedures

Functions and procedures, together called routines, are subprograms which perform certain tasks on the basis of a number of input parameters. Function returns a value after execution, whereas procedure does not. mikroBasic does not support inline routines.

Functions

Functions are declared as follows:

```
sub function function_name(parameter_list) as return_type
  [ local declarations]
   function body
end sub
```

The **function_name** specifier represents a function name and can be any valid identifier. The **parameter_list** specifier within parenthesis represents a list of formal parameters being declared similar to variables. In order to pass a parameter by address to a function, it is necessary to add the **byref** keyword at the beginning of parameter declaration. Local declarations are optional declarations of variables, constants and labels and refer to the given function only. A function body represents a sequence of statements to be executed upon calling the function. The **return_type** specifier represents the type of a function return value which can be of complex type. Example below illustrates how to define and use a function returning a complex type.

```
structure TCircle             ' Structure
  dim CenterX, CenterY as word
  dim Radius as byte
end structure

dim MyCircle as TCircle       ' Global variable

' DefineCircle function returns a Structure
sub function DefineCircle(dim x, y as word, dim r as byte) as TCircle
  result.CenterX = x
  result.CenterY = y
  result.Radius = r
end sub

main:
' Get a Structure via function call
  MyCircle = DefineCircle(100, 200, 30)

' Access a Structure field via function call
  MyCircle.CenterX = DefineCircle(100, 200, 30).CenterX + 20
'                    |----------------------| |------|
'                       |                        |
'                    Function returns TCircle     Access to one field of TCircle
end.
```

Calling function

A function is called by specifying its name followed by actual parameters placed in the same order as their matching formal parameters. The compiler is able to make mismatching parameters to get appropriate type according to implicit conversion rules. If there is a function call in an expression, the function return value will be used as an operand in that expression. Here's a simple function which calculates x^n on the basis of input parameters x and n (n > 0):

```
sub function power(dim x, n as byte) as longint
dim i as byte
  i = 0
  result = 1
  if n > 0 then
    for i = 1 to n
      result = result*x
    next i
  end if
end sub
```

By calling this function, it is possible to calculate, for example, 3^{12}:

Procedures

Procedures are declared as follows:

```
tmp = power(3, 12)
```

The **procedure_name** specifier represents a procedure name and can be any valid identifier. The **parameter_list** specifier within parentheses represents a list of formal parameters which are declared similar to variables.

```
sub procedure procedure_name(parameter_list)
  [local declarations]
  procedure body
end sub
```

In order to pass a parameter by address to a procedure, it is necessary to add the **byref** keyword at the beginning of the parameter declaration. Local declarations are optional declarations of variables, constants and labels and refer to the given procedure only. A procedure body represents a sequence of statements to be executed upon calling the procedure.

Calling procedure

A procedure is called by specifying its name followed by actual parameters placed in the same order as their matching formal parameters.

Operators

Operators are tokens denoting operations to be performed upon operands in an expression. If the order of execution is not explicitly determined using parentheses, it will be determined by the operator precedence. There are 4 precedence categories in mikroBasic. Operators in the same category have equal precedence. Each category has associativity rules, either left-to-right (→) or right-to-left (←). In the absence of parentheses, these rules resolve grouping of expressions with operators of equal precedence.

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

Arithmetic Operators

Arithmetic operators are used for performing computing operations. Operands of **char** type are bytes and can be used as unsigned operands in arithmetic operations therefore. All arithmetic operators associate from left to right.

Operator	Operation	Operands	Result
+	addition	byte, short, integer, word, longint, longword, real	byte, short, integer, word, longint, longword, real
-	subtraction	byte, short, integer, word, longint, longword, real	byte, short, integer, word, longint, longword, real
*	multiplication	byte, short, integer, word, longword, real	byte, integer, word, longint, longword, short, real
/	Division of floating-point objects	byte, short, integer, word, longword, real	byte, short, integer, word, real
div	Division and rounding down to the nearest integer	byte, short, integer, word, longint, longword	byte, short, integer, word, longint, longword
mod	Modulus operator returns the remainder of integer division (cannot be used with floating-point objects)	byte, short, integer, longint, word, longword	byte, short, integer, longint, word, longword

Division by Zero

If a zero (0) is used explicitly as the second operand in the division operation (x div 0), the compiler will report an error and will not generate a code. In case of implicit division where the second operand is an object the value of which is 0 (x div y, where y=0), the result will be undefined.

Unary Arithmetic Operators

The '-' operator can be used as a prefix unary operator to change the sign of an object. The '+' operator can also be used as a unary arithmetic operator, but it doesn't affect the object. For example:

```
b = -a
```

Relational Operators

Relational operators are used in logic operations. All relational operators return TRUE or FALSE and associate from left to right.

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

Relational Operators in Expressions

The precedence of arithmetic and relational operators enables complex expressions without parentheses to generate expected results. For example:

```
a + 5 >= c - 1.0 / e    ' ➔ (a + 5) >= (c - (1.0 / e))
```

Bitwise Operators

Bitwise operators are used for modifying individual bits of an operand. Bitwise operators associate from left to right. The only exception is the bitwise complement operator **not** which associates from right to left.

Bitwise Operators Overview

Operator	Operation
and	AND operator compares pairs of bits and generates 1 if both bits are 1, otherwise it generates 0.
or	OR operator compares pairs of bits and generates 1 if either or both bits are 1, otherwise it generates 0.
xor	XOR operator compares pairs of bits and generates 1 if the bits are complementary, otherwise it generates 0.
not	Bitwise complement (unary) inverts each bit.
<<	Shift left operator moves bits to the left, discards the leftmost bit and assigns 0 to the rightmost bit.
>>	Shift right operator moves bits to the right and discards the rightmost bit. If the object is unsigned, the leftmost bit is assigned 0. Otherwise, it is assigned a bit sign.

Logical Bitwise Operations

Bitwise operators **and**, **or** and **xor** perform logical operations upon appropriate bit pairs of their operands. The **not** operator complements each bit of an operand. For example:

```
$1234 and $5678                    ' equals $1230 because ..
' 1234 : 0001 0010 0011 0100
' 5678 : 0101 0110 0111 1000
' --------------------------
' and  : 0001 0010 0011 0000       (that is, $1230)
```

Bitwise Shift Operators

There are two shift operators in mikroBasic. These are the **<<** operator which moves bits to the left and the **>>** operator which moves bits to the right. Both operators have two operands each. The left operand is an object to move, whereas the right operand is a number of positions to move the object by. Both operands must be of **integral** type. The right operand must be a positive value. By shifting an operand left (**<<**), the leftmost bits are discarded, whereas 'new' bits on the right are assigned zeroes. Accordingly, shifting unsigned operand to the left by n positions is equivalent to multiplying it with 2^n if all discarded bits are zeros. The same applies to signed operands if all discarded bits are equal to the sign bit. By shifting operand right (**>>**), the rightmost bits are discarded, whereas 'new' bits on the left are assigned zeroes (in case of unsigned operand) or the sign bit (in case of signed operand). Shifting operand to the right by n positions is equivalent to dividing it by 2^n.

Expressions

An expression is a sequence of operators, operands and punctuators that returns a value. Primary expressions include literals, constants, variables and function calls. These can be used for creating more complex expressions by means of operators. The way operands and subexpressions are grouped does not necessarily represent the order in which they are evaluated in mikroBasic.

Statements

Statements define operations within a program. In the absence of jump and selection statements, statements are executed sequentially in the order of appearance in the program code.

Assignment Statements

Assignment statements look as follows:

```
variable = expression
```

The assignment statement evaluates the expression and assigns its value to a variable by applying all implicit conversion rules. The **variable** specifier can be any declared variable, whereas **expression** represents an expression the value of which matches the given variable. Do not confuse the assignment operator '=' with relational operator '=' used for testing equality.

Conditional Statements

Conditional or selection statements make selection from alternative courses of program execution by testing certain values.

If Statement

If statement is a conditional statement. The syntax of the **if** statement looks as follows:

```
if expression then
  statement1
[else
  statement2]
end if
```

If **expression** is true, **statement1** executes. Otherwise, **statement2** executes. The **else** keyword with an alternate statement (**statement2**) is optional.

Select Case Statement

The **select case** statement is a conditional statement of multiple branching. It consists of a control statement (selector) and a list of possible values of that expression. The syntax of the **select case** statement is:

```
select case selector
  case value_1
    statement_1
  ...
  case value_n
     statement_n
  [case else
    default_statement]
end select
```

The **selector** specifier is a control statement evaluated as integral value. Specifiers **value1..value_n** represent selector's possible values and can be literals, constants or expressions. Specifiers **statement1..statement_n** represent statements. The **else** clause is optional. First, the selector value is evaluated. It is then compared to all available values. If the value match is found, the appropriate statement will be executed, and the **select case** statement terminates. In the event that there are multiple matches, the first matching statement will be executed. If none of the values matches the selector, then the **default_statement** statement in the **else** clause (if there is one) is executed.
Here's a simple example of the **select case** statement:

```
select case operator
  case "*"
    res = n1 * n2
  case "/"
    res = n1 / n2
  case "+"
    res = n1 + n2
  case "-"
    res = n1 - n2
  case else
    res = 0
    cnt = cnt + 1
end select
```

Possible values of the control statement can also be grouped so that a few values refer to a single statement. It is just necessary to name all the values and separate them by commas:

```
select case reg
  case 0
    opmode = 0
  case 1,2,3,4
    opmode = 1
  case 5,6,7
    opmode = 2
end select
```

Nested Case Statements

Case statements can also be defined within another **case** statements. As mentioned before, the process is referred to as nesting.

Iteration Statements

Iteration statements enable a set of statements to be looped. Statements **break** and **continue** can be used for controlling the flow of loop execution. The **break** statement terminates the loop in which it exists, while **continue** starts new iteration of the loop.

For Statement

The **for** statement implements an iterative loop when the number of iterations is specified. The syntax of the **for** statement is as follows:

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

The **counter** specifier is a variable which increments by the step value (**step_value**) with each iteration of the loop. The **step_value** parameter is an optional integral value and defaults to 1 if omitted. Before the first iteration, the counter is set to initial value (**initial_value**) and will increment (or decrement) until it reaches the final value (**final_value**). The given statement (**statement**) will be executed with each iteration. It can be any statement which doesn't change the value of the **counter** variable. The **initial_value** and **final_value** should be expressions compatible with the **counter** variable, whereas the **statement** specifier can be any statement that does not change the counter value. The **step_value** parameter can be a negative value, thus enabling countdown. Here is an example of calculating scalar product of two vectors, **a** and **b**, of length **n**, using the **for** statement:

```
s = 0
for i = 0 to n-1
  s = s + a [ i] * b [ i]
next i
```

Endless Loop

The **for** statement results in an endless loop if the **final_value** equals or exceeds the range of the **counter** variable. Here is an example of an endless loop, as the **counter** variable can never reach the value 300:

```
dim counter as byte
...
for counter = 0 to 300
  nop
next counter
```

Another way of creating an endless loop in mikroBasic is by means of the **while** statement.

While Statement

The **while** statement implements an iterative loop when the number of iterations is not specified. It is necessary to check the iteration condition before loop execution. The syntax of the **while** statement is as follows:

```
while expression
  statement
wend
```

The **statement** statement is executed repeatedly as long as the value of the **expression** expression is true. The expression value is checked before the next iteration is executed. Thus, if the expression value is false before entering the loop, no iteration executes, i.e. the **statement** statement will never be executed. Probably the easiest way of creating an endless loop is by using the statement:

```
while TRUE
  ...
wend
```

Do Statement

The **do** statement implements an iterative loop when the number of iterations is not specified. The statement is executed repeatedly until the expression evaluates true. The syntax of the **do** statement is as follows:

```
do
  statement
loop until expression
```

The **statement** statement is iterated until the **expression** expression becomes true. The expression is evaluated after each iteration, so the loop will execute the statement at least once. Here is an example of calculating scalar product of two vectors, using the **do** statement:

```
s = 0
i = 0
...
do
  s = s + a[i] * b[i]
  i = i + 1
loop until i = n
```

Jump Statements

mikroBasic supports the following jump statements: **break**, **continue**, **exit**, **goto** and **gosub**.

Break Statement

Sometimes it is necessary to stop the loop from within its body. The **break** statement within loop is used to pass control to the first statement following the respective loop. For example:

```
' Wait for CF card to be inserted;
Lcd_Out(1, 1, "No card inserted")
while true
  if Cf_Detect() = 1 then
    break
  end if
  Delay_ms(1000)
wend

' CF card is inserted ...
Lcd_Out(1, 1, "Card detected")
```

Continue Statement

The **continue** statement within the loop is used for starting new iteration of the loop. Statements following the **continue** statement will not be executed.

```
' continue jumps here
for i = ...
    ...
    continue
    ...
  next i
```

```
' continue jumps here
while condition
    ...
    continue
    ...
  wend
```

```
do
    ...
    continue
    ...
' continue jumps here
loop until condition
```

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. Here is a simple example:

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

Goto Statement

The **goto** statement is used for executing an unconditional jump to appropriate part of the program. The syntax of the **goto** statement is:

```
goto label_name
```

This statement executes a jump to the **label_name** label. It is not possible to jump into or out of a procedure or function. The **goto** statement can be used for breaking out of any level of nested structures. It is not advisable to jump into a loop or other structured statement, as it may give unexpected results. The use of the **goto** statement is generally discouraged as practically every algorithm can be realized without it, resulting in legible structured programs. However, the **goto** statement is useful for breaking out of deeply nested control structures:

```
for i = 0 to n
  for j = 0 to m
    ...
    if disaster
      goto Error
    end if
    ...
  next j
next i
 ...
Error: ' error handling code
```

Gosub Statement

The **gosub** statement provides unconditional jump to the specified destination in the program:

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

It is used for executing jump to the **label_name** label. When the **return** statement is reached, the program will proceed with execution from the statement following the **gosup** statement. It may appear either before or after the label declaration in the program.

asm Statement

mikroBasic allows embedding assembly instruction in the program code by means of the **asm** statement. Assembly instructions may be grouped together using the **asm** keyword:

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

mikroBasic comments are allowed in embedded assembly code. Besides, single-line assembly comments may be written using a semicolon ';' before the comment.

```
program test
 dim myvar as word
 main:
 myvar = 0
 asm
  MOVLW 10
  MOVWF _myvar
 end asm
end.
```

Directives

Directives are words of special significance which provide additional possibilities when compiling and showing results.

Compiler Directives

mikroBasic treats comments starting with a '#' sign similar to compiler directives. Such directives, among other things, enable the program code to be conditionally compiled, i.e. it selects particular sections of the code to be compiled. All compiler directives must be completed within the file in which they have begun.

Directives $DEFINE and $UNDEFINE

The **$DEFINE** directive is used for defining a conditional compiler constant (flag). The flag can be any valid identifier. Flags have separate name space so that no confusions over program identifiers are possible. Only one flag per directive can be defined. For example:

```
$DEFINE extended_format
```

The **$UNDEFINE** directive is used for undefining ("clearing") previously defined flag.

Directives $IFDEF..$ELSE

Conditional compilation is carried out using the **$IFDEF** directive. It tests whether a flag is currently defined or not (using the **$DEFINE** directive). The **$IFDEF** directive is terminated by the **$ENDIF** directive and may have an optional **$ELSE** clause:

```
#IFDEF flag
  block of code
#ENDIF
#IFDEF flag_n
  block of code n ]
[  #ELSE
  alternate block of code ]
#ENDIF
```

First, **$IFDEF** checks if the flag is defined or not. If so, only **<block of code>** will be compiled. Otherwise, **<alternate block of code>** will be compiled. **$ENDIF** ends the conditional compilation sequence. The result of the preceding scenario is that only one section of the code (possibly empty) will be compiled. This section may contain additional (nested) conditional clauses. Each **$IFDEF** directive must be terminated with **$ENDIF**. Here is an example:

```
' Uncomment the appropriate flag:
'#DEFINE resolution8
 #IFDEF resolution8 THEN
  ... ' code specific to 8-bit resolution
 #ELSE
  ... ' default code
 #ENDIF
```

Predefined Flags

mikroBasic has predefined flags which can be used for compiling program code for different hardware platforms.

Linker Directives

mikroBasic uses internal algorithm for the purpose of distributing objects within memory. If it is necessary to have a variable or a routine at some specific predefined address, linker directives **absolute**, **org** and **orgall** must be used.

Directive absolute

The **absolute** directive specifies the starting variable address in RAM. For multi-byte variables, the higher bytes will be stored at adjacent, consecutive locations starting from the given location. This directive is appended to variable declaration:

```
dim x as byte absolute $22
' Variable x will occupy 1 byte at address $22

dim y as word absolute $23
' Variable y will occupy 2 bytes at addresses $23 and $24
```

Directive org

The **org** directive specifies the starting address of a routine in ROM. It is appended to the routine declaration. For example:

```
sub procedure proc(dim par as byte) org $200
' Procedure proc will start at address $200
...
end sub
```

Constant aggregates (records, arrays) can also be allocated at the specified address in ROM by means of the **org** directive.

```
const arr as byte[ 10] = (0,1,2,3,4,5,6,7,8,9) org 0x400
' const array will occupy 10 bytes at address 0x400
```

Directive orgall

The **orgall** directive is used for specifying the starting address of a routine in ROM from where placing of all routines and constants starts. For example:

```
main:
  orgall(0x200) ' All routines and constants in the program will be
stored above the 0x200 address, including this address as well.

  ...

end.
```

If you want to learn more about our products, please visit our website: www.mikroe.com

If you are experiencing some problems with any of our products or just need additional information, please place your ticket at : www.mikroe.com/en/support

If you have any question, comment or business proposal, do not hesitate to contact us: office@mikroe.com