QUICK REFERENCE GUIDE FOR mikroC

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

Lexical Elements Overview

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

Whitespace

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

```
char i;
unsigned int j;
```

and

```
char
i ;

    unsigned int j;
```

are lexically equivalent and parse identically giving nine tokens each:

```
char
i
;
unsigned
int
j
;
```

Whitespace in Strings

A whitespace may occur within string literals. In this case it is not used as a delimiter, but as a common character, i.e. represents part of the string. 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"
;
```

Tokens

A token is the smallest element of the mikroC programming language which the compiler recognizes. 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 standard mikroC keywords, there is a set of predefined identifiers (of constants and variables) referred to as reserved words. They describe a specific microcontroller and cannot be redefined. Here is a list of the mikroC keywords in alphabetical order:

__asm	far*	static	catch **
_asm	fdata*	struct	cdecl **
absolute	float	switch	inline **
asm	for	typedef	throw **
at	goto	union	true **
auto	idata*	unsigned	try **
bdata*	if	using	class **
bit	ilevel*	void	explicit **
break	int	volatile	friend **
case	io*	while	mutable **
cdata*	large*	xdata*	namespace **
char	long	ydata*	operator **
code*	ndata*	__automated **	private **
compact*	near*	__cdecl **	protected **
const	org	__export **	public **
continue	pascal	__finally **	template **
data*	pdata*	__import **	this **
default	register	__pascal **	typeid **
delete	return	__stdcall **	typename **
dma*	rx*	__try **	virtual **
do	sbit	_cdecl **	
double	sfr*	_export **	* architecture
else	short	_import **	dependent
enum	signed	_pasclal **	
extern	sizeof	_stdcall **	** reserved for
false	small*	bool **	future upgrade

Comments

A comment is a part of the program used to clarify program operation or to provide more information about it. Comments are intended for programmers' use only and are removed from the program before compiling. There are single-line and multi-line comments in mikroC. Multi-line comments are enclosed in /* and */ as in example below:

```
/* Type your comment here. It may span multiple lines. */
```

Single-line comments start with '//'. Here is an example:

```
// Type your comment here.
// It may span one line only.
```

Also, blocks of assembly instructions may introduce single-line comments by placing ';' in front of the comment:

```
asm {
  some_asm ; This assembly instruction …
}
```

Identifiers

Identifiers are arbitrary names used to designate the basic language objects such as labels, types, constants, variables and functions. Identifiers may contain all the letters of alphabet (both upper and lower case), the underscore character '_' as well as digits from 0 to 9. The first character of an identifier must be a letter or an underscore. mikroC is case sensitive by default, which means that Sum, sum and suM are not considered equivalent identifiers. The case sensitivity feature can be enabled or disabled using the appropriate option within mikroC IDE. Yet identifier names are arbitrary (within the stated rules), 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
if            // NO -- cannot match reserved word
j23.07.04     // NO -- cannot contain special characters (dot)
```

Literals

Literals are tokens which represent 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.

Integral Literals

Integral literals can be written in hexadecimal (base 16), decimal (base 10), octal (base 8) or binary (base 2) notation.

- Integral literals with prefix 0x (or 0X) are considered hexadecimal numbers. For example, 0x8F.
- In decimal notation, integral literals are represented as a sequence of digits (without commas, spaces or dots) with optional prefix + or -. Integral literals without prefix are considered positive. Accordingly, number 6258 is equivalent to number +6258.
- Integral literals starting with zero are considered octal numbers. For example, 0357.
- Integral literals strating with 0b (or 0B) are considered binary numbers. For example, 0b10100101.

Here are some examples of integral literals:

```
0x11    // hex literal equal to decimal 17
11      // decimal literal
011     // octal literal equal to decimal 9
0b11    // binary literal equal to decimal 3
```

The allowed range for integral literals is defined by the *long* type for signed literals and the *unsigned long* type for unsigned literals.

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

A character literal is an ASCII character, enclosed in single quotation marks.

String Literals

A string literal represents a sequence of ASCII characters enclosed in double 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 final 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 double quotation marks (null string) is stored as a single null character. Here are a few examples of 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
```

Escape Sequences

A backslash character '\' is used to introduce an escape sequence, which allows a visual representation of some non-printing characters. One of the most commonly used escape constants is a newline character '\n'. A backslash is used with an octal or hexadecimal value to represent its ASCII symbol. Table on the right shows various forms of escape sequences.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (Linefeed)
\r	0x0D	CR	Carriage Return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical Tab
\\	0x5C	\	Backslash
\'	0x27	'	Single quote (Apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O = string of up to 3 octal digits
\xH		any	H = string of hex digits
\XH		any	H = string of hex digits

Disambiguation

Some ambiguation may arise when using escape sequences like in the example below. This literal string is to be interpreted as '\x09' and '1.0 Intro'. However, mikroC compiles it as \x091 and '.0 Intro'.

```
Lcd_Out_Cp("\x091.0 Intro");
```

To avoid such problems, the code should be rewritten as follows:

```
Lcd_Out_Cp("\x09" "1.0 Intro")
```

Line Continuation with Backslash

A backslash '\' may be used as a continuation character to extend a string constant across line boundaries:

```
"This is really \
        a one-line string."
```

Enumeration Constants

Enumeration constants are identifiers defined in *enum* type declarations. For example:

```
enum weekdays { SUN = 0, MON, TUE, WED, THU, FRI, SAT };
```

Identifiers (enumerators) that are being used must be unique within the scope of the enum declaration. Negative initializers are allowed.

Punctuators

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

- [ ] - square brackets;
- ( ) - parentheses;
- { } - braces
- , - comma;
- ; - semicolon;
- : - colon
- . - dot
- * - asterisk
- = - equal sign
- # - pound sign

Most of these punctuators also function as operators.

Square brackets

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

```
char ch, str[] = "mikro";
int mat[3][4]; /* 3 x 4 matrix */
ch = str[3];   /* 4th element */
```

Parentheses

Parentheses ( ) are used to group expressions, isolate conditional expressions and indicate function calls and function declarations:

```
d = c * (a + b);        /* override normal precedence */
if (d == z) ++x;        /* essential with conditional statement */
func();                 /* function call, no args */
void func2(int n);      /* function declaration with parameters */
```

Braces
Braces { } indicate the start and end of a compound statement:

```
if (d == z) {
  ++x;
  func();
}
```

A closing brace serves as a terminator for compound statements, so that a semicolon is not required after the closing brace '}', except in structure declarations.

Comma
Commas ',' are used to separate parameters in function calls, identifiers in declarations and elements of initializer:

```
Lcd_Out(1, 1, txt);
char i, j, k;
const char MONTHS[12] = (31,28,31,30,31,30,31,31,30,31,30,31);
```

Semicolon
A semicolon ';' is a statement terminator. Any legal C expression (including the empty one) followed by a semicolon is interpreted as a statement, known as expression statement. A semicolon is also used for denoting the place in assembly blocks where comments start.

```
a + b;     /* Evaluate a + b, but discard value */
++a;       /* Side effect on a, but discard value of ++a */
;          /* Empty expression, or a null statement */
```

Colon
A colon ':' is used to indicate a label:

```
start:  x = 0;
  ...
goto start;
```

Dot
A dot '.' is used to indicate the access to a structure or union field. It can also be used for accessing individual bits of registers in mikroC. For example:

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

Asterisk (Pointer Declaration)
An asterisk '*' in a variable declaration denotes the creation of a pointer to a type. Pointers with multiple levels of indirection can be declared by indicating an appropriate number of asterisks:

```
char *char_ptr;  /* a pointer to char is declared */
```

Equal Sign

An equal sign '=' is used to separate variable declarations from initialization lists as well as the left and right side in assignments expressions.

```
int test[ 5]  = {  1, 2, 3, 4, 5 };
int x = 5;
```

Pound Sign (Preprocessor Directive)

Pound sign '#' indicates a preprocessor directive when it appears as the first nonwhite-space character in a line.

Declarations

Declarations introduce one or several names to the program and let the compiler know what the name represents, what type it is, what operations are allowed upon it, etc. This section describes concepts related to declarations, definitions, declaration specifiers and initialization. Here is a list of objects that can be declared:

Variables;
Constants;
Functions;
Types;
Structure, union and enumeration tags;
Structure members;
Union members;
Arrays of other types;
Statement labels; and
Preprocessor macros.

Declarations and Definitions

Defining declarations, also known as definitions, introduce the name of an object and create an object as well. There may be many declarations for the same identifier, especially in a multifile program, whereas only one definition is allowed. For example:

```
/* Definition of variable i: */
int i;

/* Following line is an error, i is already defined! */
int i;
```

Declarations and Declarators

A declaration contains specifier(s) followed by one or more identifiers (declarators). It begins with optional storage class specifiers, type specifiers or other modifiers. Identifiers are separated by commas and the declaration is terminated by a semicolon:

```
storage-class [ type-qualifier]  type var1 [ =init1], var2 [ =init2], ... ;
```

Storage Classes

A storage class defines the scope (visibility) and lifetime of variables and/or functions within a C program. The storage classes that can be used in a C program are auto, register, static and extern.

Auto

The *auto* storage-class specifier declares an automatic variable (a variable with a local lifetime). An auto variable is visible only within the block in which it is declared. The *auto* storage-class specifier can only be applied to names of variables declared in a block or to names of function parameters. However, these names have automatic storage by default. Therefore the *auto* storage class specifier is usually redundant in a data declaration.

Register

The *register* storage-class specifier is used to define local variables that should be stored in a register instead of RAM. At the moment this modifier has no special meaning in mikroC. MikroC simply ignores requests for register allocation.

Static

The *static* storage class specifier lets you define variables or functions with internal linkage, which means that each instance of a particular identifier represents the same variable or function within one file only. In addition, variables declared *static* have static storage duration, which means that memory for these variables is allocated when the program begins running and is freed when the program terminates. Static storage duration for a variable is different from file or global scope. A variable can have static duration, but local scope.

Extern

The *extern* storage class specifier lets you declare objects that can be used in several source files. An extern declaration makes a described variable usable by the succeeding part of the current source file. This declaration does not replace the definition. It is used to describe a variable that is externally defined.

An extern declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword extern is optional.

If a declaration for an identifier already exists within the file scope, any extern declaration of the same identifier found within a block refers to the same object. If no other declaration for the identifier exists within the file scope, the identifier has external linkage.

Type Qualifiers

C type qualifiers add special properties to the variables being declared. mikroC provides two keywords: const and volatile.

Const

The *const* qualifier is used to indicate that variable value cannot be changed. Its value is set at initialization.

Volatile

The *volatile* qualifier indicates that variable values can be changed both with or without user's interference in the program. The compiler should not optimize such variable.

Typedef Specifier

The *typedef* declaration introduces a name that, within its scope, becomes a synonym for the specified type:

```
typedef <type_declaration> synonym;
```

You can use *typedef* declarations to construct shorter or more meaningful names for types already defined by the language or declared by the user. *Typedef* names allow you to encapsulate implementation details that may change.

Unlike the *class, struct, union*, and *enum* declarations, the *typedef* declarations do not introduce new types, but new names for existing types.

asm Declaration

mikroC allows assembly instructions to be embedded in the source code by means of the asm declaration:

```
asm {
  block of assembly instructions
}
```

A single assembly instruction can be embedded in the C code:

```
asm assembly_instruction;
```

Initialization

The initial value of a declared object can be set at the time of declaration. This procedure is called initialization. For example:

```
int i = 1;
char *s = "hello";
struct complex c = {0.1, -0.2};
// where 'complex' is a structure containing two objects of float type
```

Scope and Visibility

Scope

The scope of an identifier is a 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.

Visibility

Likewise, the visibility of an identifier is a 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

mikroC is strictly typed language, which means that each object, function and expression must have its type defined before the compiling process starts. Type checking may prevent objects from being illegally assigned or accessed.

mikroC supports standard (predefined) data types such as signed and unsigned integrals of various sizes, arrays, pointers, structures and unions etc. Also, the user can define new data types using the *typedef* specifier. For example:

```
typedef char MyType1[10];
typedef int MyType2;
typedef unsigned int * MyType3;
typedef MyType1 * MyType4;

MyType2 mynumber;
```

Arithmetic Types

Arithmetic type specifiers include the following keywords: *void, char, int, float* and *double*, preceeded by the following prefixes: *short, long, signed* and *unsigned*. There are *integral* and *floating point* arithmetic types.

Integral Types

Types char and int, along with their various forms, are considered to be integral data types. Their forms are created using one of the following prefix modifiers: *short, long, signed* and *unsigned*. The table below gives an overview of integral data types. Keywords in parentheses may be (and usually are) omitted.

Type	Size in bytes	Range
(unsigned) char	1	0 .. 255
signed char	1	- 128 .. 127
(signed) short (int)	1	- 128 .. 127
unsigned short (int)	1	0 .. 255
(signed) int	2	-32768 .. 32767
unsigned (int)	2	0 .. 65535
(signed) long (int)	4	-2147483648 .. 2147483647
unsigned long (int)	4	0 .. 4294967295

Floating-point Types

Types *float* and *double*, along with the long double form, are considered to be *floating-point* types. An overview of the *floating-point* types is given in the table below:

Type	Size in bytes	Range
float	4	$-1.5 * 10^{45} .. +3.4 * 10^{38}$
double	4	$-1.5 * 10^{45} .. +3.4 * 10^{38}$
long double	4	$-1.5 * 10^{45} .. +3.4 * 10^{38}$

Enumerations

The *enumeration* data type is used for representing an abstract, discreet set of values with the appropriate symbolic names. An enumeration is declared as follows:

```
enum colors { black, red, green, blue, violet, white } c;
```

Void Type

void is a special type indicating the absence of value. There are no objects of void type. Instead, the void type is used for deriving more complex types.

```
void print_temp(char temp) {
  Lcd_Out_Cp("Temperature:");
  Lcd_Out_Cp(temp);
  Lcd_Chr_Cp(223);  // degree character
  Lcd_Chr_Cp('C');
}
```

Pointers

A pointer is a variable which holds memory address of an object. While the variable directly accesses the memory address, the pointer can be thought of as a reference to that address. Pointers are declared in a similar manner to any other variable, but with an asterisk '*' ahead of the identifier. Pointers must be initializied prior to being used. For example, to declare a pointer to an object of *int* type, it is necessary to write:

```
int *pointer_name;    /* Uninitialized pointer */
```

To access data stored at a memory location aimed to by the pointer, it is necessary to add an asterisk prefix '*' to the pointer name. For example, to declare variable *p* which points to an object of *unsigned int* type, and to assign value 5 to the object, do as follows:

```
unsigned int *p;
...
*p = 5;
```

A pointer can be assigned to another pointer of compatible type. It makes both pointers aim at the same memory location. Modifying the object pointed to by one pointer causes the object pointed to, by the other pointer, to change automatically as they share the same memory location.

Null Pointers

A null pointer has a reserved value which is often but not necessarily the value zero, indicating that it refers to no object. To assign integer constant 0 to a pointer means to assign a null pointer value to it.

```
int *pn = 0;       /* Here's one null pointer */

/* We can test the pointer like this: */
if ( pn == 0 ) {  ... }
```

The pointer to void must not be confused with the null pointer. The following declaration declares vp as a generic pointer capable of being assigned to by any pointer value, including null.

```
void *vp;
```

Function Pointers

As their name indicates, function pointers are pointers which aim at a function.

```
int addC(char x,char y){
   return x+y;
}

int subC(char x,char y){
   return x-y;
}

int mulC(char x,char y){
   return x*y;
}


int divC(char x,char y){
   return x/y;
}


int modC(char x,char y){
   return x%y;
}

//array of pointer to functions that receive two chars and returns int
int   (*arrpf[])(char,char) = {addC, subC, mulC, divC, modC};

int res;
char i;
void main() {

    for (i=0;i<5;i++){
      res = arrpf[i](10,20);
    }
}
```

Structures

A structure is a derived type which usually represents a user-defined collection of named members (or components). These members can be of any type.

Structure Declaration and Initialization

Structures are declared using the struct keyword:

```
struct tag {member-declarator-list};
```

Here, *tag* is the name of a structure; *member-declarator-list* is a list of structure members, i.e. list of variable declarations. The variables of structured type are declared in the same manner as the variables of any other type.

Incomplete Declarations

Incomplete declarations are also known as forward declarations. A pointer to one structure type can legally appear in the declaration of another structure type even before the pointed structure type is defined:

```
struct A;  // incomplete
struct B {struct A *pa;};
struct A {struct B *pb;};
```

The first appearance of A is called incomplete because there is no definition thereof at that point. An incomplete declaration is allowed here as the definition of B doesn't require the size of A.

Untagged Structures and Typedefs

If the structure tag is omitted, an untagged structure is created. The untagged structures can be used to declare identifiers in the comma-delimited member-declarator-list to be of the given structure type (or derived from it), but additional objects of this type cannot be declared elsewhere. It is possible to create a typedef while declaring a structure, with or without tag:

```
/* With tag: */
typedef struct mystruct {  ... } Mystruct;
Mystruct s, *ps, arrs[10];  /* same as struct mystruct s, etc. */

/* Without tag: */
typedef struct {  ... } Mystruct;
Mystruct s, *ps, arrs[10];
```

Structure Size

The structure size in memory can be retrieved by means of the sizeof operator. The structure size is not necessarily equal to the summed up sizes of its members. It is often greater due to certain limitations of memory storage.

Assignment

Variables of the same structured type may be assigned to each other by means of a simple assignment operator '='. It makes the entire contents of a variable to be copied to destination, regardless of the inner complexity of the given structure. Note that two variables are of the same structured type only if they are both defined by the same declaration or using the same type identifier. For example:

```
/* a and b are of the same type: */
struct {int m1, m2;} a, b;

/* But c and d are _not_ of the same type although
their structure descriptions are identical: */
struct {int m1, m2;} c;
struct {int m1, m2;} d;
```

Structure Member Access

Structure and union members can be accessed using either of the two following selection operators:

- . (period)
- -> (right arrow).

Operator '.' is called a direct member selector and is used to directly access one of the structure members.

```
s.m   // direct access to member m
```

Operator '->' is called indirect (or pointer) member selector.

```
ps->m     // indirect access to member m;
          // identical to (*ps).m
```

Union Types

Union types are derived types sharing many syntactic and functional features with structure types. The main difference is that union members share the same memory space.

Union Declaration

Unions have the same declaration as structures except the union keyword used instead of struct:

```
union tag { member-declarator-list };
```

Bit Fields

Bit fields represent a specified number of bits that may or may not have an associated identifier. Bit fields offer a possibility of subdividing structures into named parts of user-defined sizes.

Bit Fields Declaration

Bit fields can be declared only in structures and unions. Declare a structure normally and assign individual fields as in the following example (fields need to be of unsigned type):

```
struct tag {
   unsigned bitfield-declarator-list;
}
```

Here, *tag* is an optional name of the structure, whereas *bitfield-declarator-list* is a list of bit fields. Each bit field identifer requires a colon and its width in bits to be explicitly specified.

Bit Fields Access

Bit fields can be accessed in the same way as structure members, i.e. by means of a direct and indirect member selector '.' and '->'.

```
// This example writes stucture Value_For_PORT0 to P0
struct myPort {
   lo_nibble :4;
   hi_nibble :4;
} Value_For_PORT0;

void main() {
   ...
   Value_For_PORT0.lo_nibble  = 7;
   Value_For_PORT0.hi_nibble  = 0x0C;
   P0 = *(char *) (void *)&Value_For_PORT0;
            // typecasting :
            // 1. address of structure to pointer to void
            // 2. pointer to void to pointer to char
            // 3. dereferencing to obtain the value
}
```

Arrays

An array is the simplest and the most commonly used structured type. A variable of array type is actually an array of objects of the same type. These objects represent elements of an array and are identified by their position in the array. An array consists of a contiguous memory locations large enough to hold all its elements.

Array Declaration

Array declaration is similar to variable declaration, with brackets following identifer:

```
element_type array_name[ constant-expression];
```

An array named as *array_name* and composed of elements of *element_type* is declared here. The *element_type* can be any scalar type (except void), *user-defined type, pointer, enumeration* or another array. The result of *constant-expression* within brackets determines a number of array elements. Here are a few examples of array declaration:

```
#define MAX = 50
int    vector_one[ 10];          /* declares an array of 10 integers */
float  vector_two[ MAX];         /* declares an array of 50 floats   */
float  vector_three[ MAX - 20]; /* declares an array of 30 floats   */
```

Array Initialization

An array can be initialized in declaration by assigning it a comma-delimited sequence of values within braces. When initializing an array in declaration, you can omit the number of elements as it will be automatically determined on the basis of the number of elements assigned. For example:

```
/* Declare an array which contains number of days in each month: */
int days[ 12]  = { 31,28,31,30,31,30,31,31,30,31,30,31};
```

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:

```
char m[ 50][ 20];    /* 2-dimensional array of size 50x20 */
```

Variable *m* represents an array of 50 elements which in turn represent arrays of 20 elements of *char* 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].
A multi-dimensional array can be initailized with an appropriate set of values within braces. For example:

```
int a[ 3][ 2]  = {{ 1,2}, { 2,6}, { 3,7}};
```

Type Conversions

The conversion is a process of changing variable type. mikroC supports both implicit and explicit types of conversion.

Implicit Conversion

Automatic conversion of a value from one data type to another by a programming language, without the programmer specifically doing so, is called implicit type conversion. The compiler preforms implicit conversion in the following situations:

- An operand is prepared for an arithmetic or logical operation;
- An assignment is made to an lvalue that has a different type than the assigned value;
- A function is provided an argument value that has a different type than the parameter;
- The value specified in the return statement of a function has a different type from the defined return type for the function.

Promotion

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

short → int, long, float, or double
char → int, long, float, or double
int → long, float, or double
long → float or double

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

```
char a;
unsigned int b;
...
a = 0xFF;
b = a;  // a is promoted to unsigned int, b equals to 0x00FF
```

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).

```
char i;
unsigned int j;
...
j = 0xFF0F;
i = j;          // i becomes 0x0F, higher byte 0xFF is lost
```

Explicit Conversion

Explicit conversion may be performed upon any expression using the unary typecast operator. A special case is conversion between signed and unsigned types. Such explicit conversion does not affect data binary representation. For example:

```
short b;
char a;
...
b = -1;
a = (char)b;     // a is 255, not 1
                 // This is because binary representation remains
                 // 11111111; it's just interpreted differently now
```

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

```
(int)b = a;   // Compiler will report an error
```

Here is an example of explicit conversion:

```
char a, b, c;
unsigned int cc;
...
a = 241;
b = 128;

c  = a + b;                  // equals 113
c  = (unsigned int)(a + b);  // equals 369
cc = a + b;                  // equals 369
```

Functions

Functions are subprograms which perform certain tasks on the basis of a number of input parameters. Functions may return a value after execution. The function return value can be used in expressions. Technically, the function call is considered to be an expression like any other. Each program must have a single function named *main* marking the entry point of the program. Functions can be declared in standard or user-supplied header files, or within program files.

Function Declaration

Functions are declared/defined in user's source files or made available by linking pre-compiled libraries. The syntax of a function declaration is:

```
result_type function_name(parameter-declarator-list);
```

function_name represents the function name and must be a valid identifier. *result_type* represents the type of function result and can be of any standard or user-defined type.

Function Prototypes

A function can be defined only once in the program, but can be declared several times, assuming that the declarations are compatible. Parameters are allowed to have different names in different declarations of the same function:

```
/* Here are two prototypes of the same function: */

int test(const char*)    /* declares function test */
int test(const char*p)   /* declares the same function test */
```

Function Definition

A function definition consists of its declaration and function body. The function body is technically a block (sequence) of local definitions and statements enclosed in braces {}. All variables declared within the function body are local to the function, i.e. they have function scope. Here is a sample function definition:

```
/* function max returns greater one of its 2 arguments: */

int max(int x, int y) {
  return (x>=y) ? x : y;
}
```

Function Calls

A function is called by specifying its name followed by an actual parameters placed in the same order as their matching formal parameters. If there is a function call in an expression, the function return value will be used as an operand in that expression.

```
function_name(expression_1, ... , expression_n);
```

Each expression in the function call is an actual argument.

Argument Conversions

The compiler is capable of making mismatching parameters get appropriate type according to implicit conversion rules.

```
int limit = 32;
char ch = 'A';
long res;

// prototype
extern long func(long par1, long par2);

main() {
   ...
   res = func(limit, ch);  // function call
}
```

Since the program has the function prototype for func, it converts limit and ch into long, using the standard rules of assignment, before passing them to appropriate formal parameter.

Ellipsis Operator

The ellipsis '...' operator consists of three successive periods with no whitespace intervening. An ellipsis can be used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types. For example:

```
void func (int n, char ch, ...);
```

This declaration indicates that func is to be defined so as that calls must have at least two arguments (int and char) but can also have any number of additional arguments.

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.

- Arithmetic operators
- Assignment operators
- Bitwise operators
- Logical operators
- Reference/Indirect operators
- Relational operators
- Structure member selectors
- Comma operator ,
- Conditional operator ?:
- Array subscript operator []
- Function call operator ()
- Sizeof operator
- Preprocessor pperators # and ##

Operators Precedence and Associativity

There are 15 precedence categories in mikroC. Operators in the same category have equal precedence. If duplicates of operators appear in the table, the operator of higher presidence is unary. Each category has either left-to-right or right-to-left associativity rule. In the absence of parentheses, these rules apply when grouping expressions with operators of equal precedence.

Precedence	Operands	Operators	Associativity
15	2	() [] . ->	→
14	1	! ~ ++ -- + - * & (type) sizeof	←
13	2	* / %	→
12	2	+ -	→
11	2	<< >>	→
10	2	< <= > >=	→
9	2	== !=	→
8	2	&	→
7	2	^	→
6	2	|	→
5	2	&&	→
4	2	||	→
3	3	?:	←
2	2	= *= /= %= += -= &= ^= |= <<= >>=	←
1	2	,	→

Arithmetic Operators

Arithmetic operators are used for performing computing operations. All arithmetic operators associate from left to right.

Operator	Operation	Precedence
	Binary Operators	
+	addition	12
-	subtraction	12
*	multiplication	13
/	division	13
%	modulus operator returns the remainder of integer division (cannot be used with floating points)	13
	Unary Operators	
+	unary plus does not affect the operand	14
-	unary minus changes the sign of the operand	14
++	increment adds one to the value of the operand. Postincrement adds one to the value of the operand after it evaluates; while preincrement adds one before it evaluates	14
--	decrement subtracts one from the value of the operand. Postdecrement subtracts one from the value of the operand after it evaluates; while predecrement subtracts one before it evaluates	14

Relational Operators

Relational operators are used in logic operations. All relational operators return either 1 (TRUE) or 0 (FALSE) and associate from left to right.

Operator	Operation	Precedence
==	equal	9
!=	not equal	9
>	greater than	10
<	less than	10
>=	greater than or equal	10
<=	less than or equal	10

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 '~' which associates from right to left.

Operator	Operation	Precedence
&	bitwise AND; compares pairs of bits and returns 1 if both bits are 1, otherwise it returns 0	8
|	bitwise (inclusive) OR; compares pairs of bits and returns 1 if either or both bits are 1, otherwise it returns 0	6
^	bitwise exclusive OR (XOR); compares pairs of bits and returns 1 if the bits are complementary, otherwise it returns 0	7
~	bitwise complement (unary); inverts each bit	14
<<	bitwise shift left; moves the bits to the left, discards the far left bit and assigns 0 to the far right bit.	11
>>	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the far left bit, otherwise sign extends	11

Logical Operators

Operands of logical operations are considered true or false, i.e. non-zero or zero, respectively. Logical operators always return either 1 or 0. Logical operators '&&' and '||' associate from left to right. The logical negation operator '!' associates from right to left.

Operator	Operation	Precedence
&&	logical AND	5
||	logical OR	4
!	logical negation	14

Conditional Operator

The conditional operator '?:' is the only ternary operator in mikroC. The syntax of the conditional operator is:

```
expression1 ? expression2 : expression3;
```

expression1 is evaluated first. If its value is true, then *expression2* evaluates and expression3 is ignored. If *expression1* evaluates to false, then *expression3* evaluates and *expression2* is ignored. The result will be the value of either *expression2* or *expression3* depending on which of them evaluates.

Simple Assignment Operator

For a common value assignment, a simple assignment operator '=' is used:

```
expression1 = expression2;
```

expression1 is an object (memory location) to which the value of *expression2* is assigned.

Compound Assignment Operators

mikroC allows more complex assignments by means of compound assignment operators. The syntax of compound assignment operators is:

```
expression1 op= expression2;
```

where op can be any of binary operators +, -, *, /, %, &, |, ^, <<, or >>.

Compound assignment above is a short form of the following expression:

```
expression1 = expression1 op expression2;
```

Bitwise Shift Operators

There are two shift operators in mikroC. 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 first operand is an object to move, whereas the second operand is a number of positions to move the object by. Both operands must be of *integral* type. The second 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 the case of unsigned operand) or the sign bit (in the case of signed operand). Shifting operand to the right by n positions is equivalent to dividing it by 2^n.

Operator	Operation	Precedence
<<	shift left	11
>>	shift right	11

Expressions

An expression is a sequence of operators, operands and punctuators that return 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 mikroC.

Comma Expressions

A special characteristic of mikroC is that it allows comma to be used as a sequence operator to form so-called comma expressions or sequences. The following sequence:

```
expression_1, expression_2, ... expression_n;
```

results in the left-to-right evaluation of each expression. The value and type of the comma expression equals to the value and type of *expression_n*.

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. Statements can be roughly divided into:

- Labeled statements;
- Expression statements;
- Selection statements;
- Iteration statements (Loops);
- Jump statements; and
- Compound statements (Blocks).

Labeled Statements

Each statement in the program can be labeled. A label is an identifier added before the statement like this:

```
label_identifier: statement;
```

The same label cannot be redefined within the same function. Labels have their own namespace and therefore label identifier can match any other identifier in the program.

Expression Statements

Any expression followed by a semicolon forms an expression statement:

```
expression;
```

A null statement is a special case of expression statement, consisting of a single semicolon ';'. A null statement is commonly used in 'empty' loops:

```
for (; *q++ = *p++ ;)
  ;  /* body of this loop is a null statement */
```

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 some_expression statement1; [ else statement2;]
```

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

Nested if statements

Nested *if* statements require additional attention. A general rule is that they are parsed starting from the innermost if statement (the most nested) and each *else* statement is bound to the nearest available *if* on its left:

```
if (expression1) statement1;
else if (expression2)
if (expression3) statement2;
else statement3;         /* this belongs to: if (expression3) */
else statement4;         /* this belongs to: if (expression2) */
```

Switch Statement

The *switch* statement is a conditional statement with multiple branching, based on a certain condition. It consists of a control expression (selector) and a list of its possible values. The syntax of the switch statement is:

```
switch (expression) {
  case constant-expression_1 : statement_1;
    ...
  case constant-expression_n : statement_n;
  [default : statement;]
}
```

First, *expression* (condition) is evaluated. The *switch* statement then compares it to all available constant-expressions following the *case* keyword. If match is found, switch passes control to that matching case causing the statement following the match to evaluate.

Iteration Statements

Iteration statements enable a set of statements to be looped.

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 ([init-expression];[condition-expression];[increment-expression]) statement;
```

Before the first iteration of the loop, *init-expression* sets variables to initial values. You cannot place declarations in *init-expression*. *Condition-expression* is checked before the first entry into the block. The for statement is executed repeatedly until the value of *condition-expression* becomes false. After each iteration of the loop, *increment-expression* increments the loop counter. Here is an example of calculating scalar product of two vectors using the *for* statement:

```
for ( s=0, i=0; i<n; i++ ) s+= a[i] * b[i];
```

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 executing the loop. The syntax of the *while* statement is as follows:

```
while (cond_exp) some_stat;
```

The *some_stat* statement is executed repeatedly as long as the value of the *cond_exp* 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 is executed. Probably the easiest way of creating an endless loop is by using the statement:

```
while (1) ...;
```

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 some_stat; while (cond_exp);
```

The *some_stat* statement is iterated until the *cond_exp* expression becomes true. The expression is evaluated after each iteration, so the loop will execute the statement at least once.

Jump Statements

mikroC supports the following jump statements: *break, continue, return* and *goto*.

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. Break is commonly used in a switch statement to stop its execution after the first positive match occurs. For example:

```
switch (state) {
  case 0: Lo();  break;
  case 1: Mid(); break;
  case 2: Hi();  break;
  default: Message("Invalid state!");
}
```

Continue Statement

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

```
while (..) {
...
if (val>0) continue;
...
// continue jumps here
}
```

```
do {
...
if (val>0) continue;
...
// continue jumps here
while (..);
```

```
for (..;..;..) {
...
if (val>0) continue;
...
// continue jumps here
}
```

Return Statement

The *return* statement is used to exit the current function optionally returning a value. The syntax is:

```
return [ expression] ;
```

This will evaluate expression and return the result. The returned value will be automatically converted into the expected function type, if needed. The expression is optional.

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_identifier ;
```

This statement executes a jump to the *label_identifier* label. The *goto* statement may occur either before or after the label declaration. The label declaration and *goto* statement must belong to the same routine. Accordingly, 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 (...) {
  for (...) {
    ...
    if (disaster)
      goto Error;
    ...
  }
}
 .
 .
 .
Error: // error handling code
```

Preprocessor

A preprocessor is an integrated text processor which prepares the source code for compiling. The preprocessor allows:

- text from a specifed file to be inserted into a certain point in the code (File Inclusion);
- specific lexical symbols to be replaced with other symbols (Macros); and
- conditional compiling which conditionally includes or omits parts of the code (Conditional Compilation).

Preprocessor Directives

Any line in the source code with a leading '#' is taken as a preprocessing directive (or a control line), unless '#' belongs to a string literal, character constant or a comment. mikroC supports standard preprocessor directives:

# (null directive)	#error	#ifndef
#define	#endif	#include
#elif	#if	#line
#else	#ifdef	#undef

File Inclusion

The *#include* directive copies a specified file into the source code. The syntax of the *#include* directive has either of the two following formats:

```
#include <file_name>
#include "file_name"
```

The preprocessor replaces the *#include* line with the content of a specified file in the source code. The placement of #include can therefore affect the scope and duration of all identifiers in the included file. The difference between these two formats lies in searching algorithm employed in trying to locate the file to be included.

Explicit Path

Placing an explicit path in *file_name*, means that only that directory will be searched. For example:

```
#include "C:\my_files\test.h"
```

Macros

Macros provide a mechanism for token replacement prior to compiling.

Defining Macros and Macro Expansions

The *#define* directive defines a macro:

```
#define macro_identifier <token_sequence>
```

Each occurrence of *macro_identifier* in the source code following this control line will be replaced by the *token_sequence* (possibly empty).

Macros with Parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) <token_sequence>
```

The optional *arg_list* is a sequence of identifiers separated by commas, like the argument list of a C function. Each comma-delimited identifier has the function of a formal argument or placeholder. Here is a simple example:

```
/* A simple macro which returns greater of its 2 arguments: */
#define _MAX(A, B) ((A) > (B)) ? (A) : (B)
```

Conditional Compilation

Conditional compilation directives are typically used for making source programs easy to change and compile in different execution environments.

Directives #if, #elif, #else, and #endif

Conditional directives *#if, #elif, #else*, and *#endif* work very similar to the common mikroC conditional statements. If the expression you write after *#if* has a nonzero value, the line group immediately following the *#if* directive is retained in the translation unit.
The syntax is:

```
#if constant_expression_1
<section_1>

[ #elif constant_expression_2
<section_2>]
   ...
[ #elif constant_expression_n
<section_n>]

[ #else
<final_section>]

#endif
```

Each *#if* directive in a source file must be matched by a closing *#endif* directive. Any number of *#elif* directives can appear between *#if* and *#endif* directives, but only one *#else* directive is allowed. The *#else* directive, if present, must precede *#endif*.

Directives #ifdef and #ifndef

The *#ifdef* and *#ifndef* directives test whether an identifier is currently defined or not. The following line:

```
#ifdef identifier
```

has exactly the same effect as *#if 1* if identifier is currently defined, or as *#if 0* if identifier is currently undefined. The *#ifndef* directive checks for the opposite condition that it is checked by *#ifdef*.

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 contact our technical support: www.mikroe.com/en/support

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