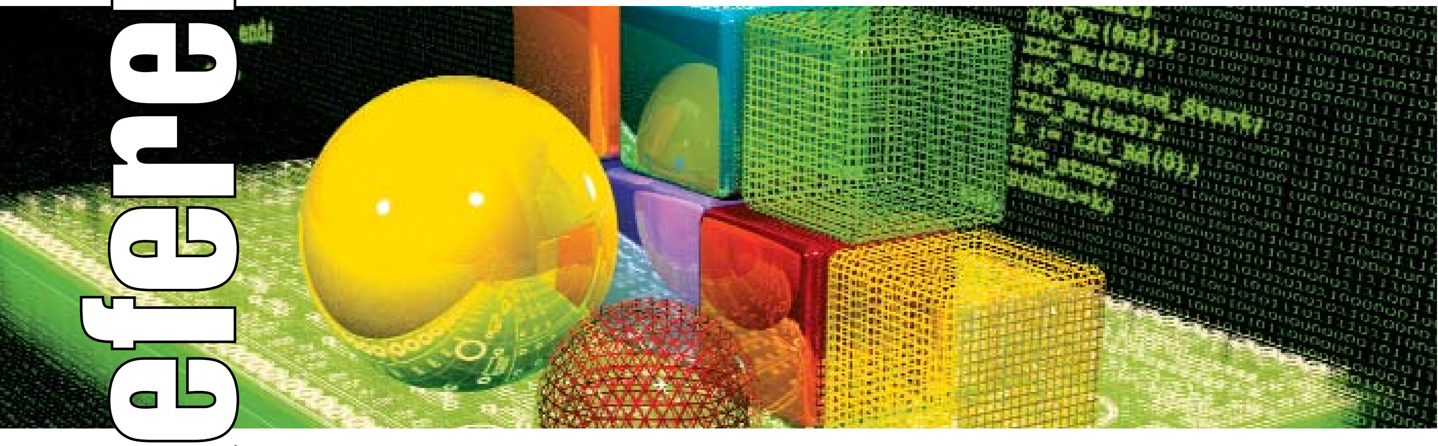
SOFTWARE AND HARDWARE SOLUTIONS FOR THE EMBEDDED WORLD

Quick Reference Guide

for C language



Quick Reference

with EXAMPLES

This reference guide is intended to quickly introduce user's to C language syntax with the aim to easily start programming micro-controllers along with other applications.

Why C in the first place? The answer is simple: C offers unmatched power and flexibility in programming microcontrollers.

Software and Hardware solutions for Embedded World

COMMENTS

C comments

C comment is any sequence of characters placed after the symbol pair /*. The comment terminates at the first occurrence of the pair */ following the initial /*.

Example:

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

C++ comments

mikroC allows single-line comments using two adjacent slashes (//).

Example:

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

CONSTANTS

Character Constants

A character constant is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-character constants have data type int.

Escape Sequences

The backslash character (\) is used to introduce an escape sequence, which allows the visual representation of certain nongraphic characters.

The following table shows the available escape sequences in mikroC:

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

String Constants

A string literal (string constant) is a sequence of any number of characters surrounded by double quotes.

Example:

```
"This is a string."
```

Enumeration Constants

Enumeration constants are identifiers defined in enum type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are of int type. They can be used in any expression where integer constants are valid.

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

KEYWORDS

```
asm         enum        signed
auto        extern      sizeof
break       float       static
case        for         struct
char        goto        switch
const       if          typedef
continue    int         union
default     long        unsigned
do          register    void
double      return      volatile
else        short       while
```

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

FUNDAMENTAL TYPES

Type	Size	Range
`(unsigned) char`	8-bit	0 .. 255
`signed char`	8-bit	- 128 .. 127
`(signed) short (int)`	8-bit	- 128 .. 127
`unsigned short (int)`	8-bit	0 .. 255
`(signed) int`	16-bit	-32768 .. 32767
`unsigned (int)`	16-bit	0 .. 65535
`(signed) long (int)`	32-bit	-2147483648 .. 2147483647
`unsigned long (int)`	32-bit	0 .. 4294967295
`float`	32-bit	±1.17549435082E-38 .. ±6.80564774407E38
`double`	32-bit	±1.17549435082E-38 .. ±6.80564774407E38
`long double`	32-bit	±1.17549435082E-38 .. ±6.80564774407E38

Enumeration
Syntax:
```
enum tag {enumeration-list};
```

This example establishes a unique integral type, `colors`, a variable `c` of this type, and a set of enumerators with constant integer values (black = 0, red = 1, ...).

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

DERIVED TYPES
ARRAYS

Array Declaration
Syntax:
```
type array_name[constant-expression];
```
Example:
```
int array_one[7];           /* an array of 7 integers */
```

Array Initialization
Example:
```
int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Multi-dimensional Arrays

Example:

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

POINTERS

Pointer Declarations

Syntax:

```
type *pointer_name;
```

Example:

```
int *p = &a;
```

or if declared like this:

```
int *p;
p = &a;
```

Now ponter p points to variable a.
`*p=5;` asignes value 5 to variable a.

Note:
You must initialize pointers before using them!.

Null Pointers

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program.

Example:

```
int *pn = 0;          /* Here's one null pointer */
int *pn = NULL;       /* This is an equivalent declaration */
```

STRUCTURES

Structure Declaration and Initialization

Syntax:

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

Example:

```
struct Dot {int x, y;};     // declaration
struct Dot p = {1, 1};      // initialization
```

Note:
The member type cannot be the same as the struct type being currently declared. However, a member can be a pointer to the structure being declared!

Structure Member Access

Example:

```
struct Dot *ptr = p;        // declares pointer to struct p
p.x = 3;                    // direct access to member x
ptr->x = 4;                 // indirect access to member x
```

`ptr->x` is identical to `(*ptr).x`!

UNIONS

Union Declaration

Syntax:

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

Difference between structure and union is that unlike structure's members, the value of only one of union's members can be stored at any time..

Union Member Access

Example:

```
union Spot {int x, y;} p;
p.x = 4;
Display(p.x);         // This is valid! Displays value of member x!
Display(p.y);         // This is invalid!
p.y = 7;
Display(p.y);         // This is valid! Displays value of member y!
```

BIT FIELDS

Bit Fields Declaration

Syntax:

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

Example:
```
struct Port {
led_0 : 1;
other_leds : 7;
} PortA;
```

Bit Fields Access

Example:
```
PortA.led_0 = 1;
PortA.other_leds = 0;
```

This code will turn ON LED0 and turn OFF rest of the leds on PORTA.

DECLARATIONS

Typedef Specifier

Syntax:
```
typedef <type-definition> synonym;
```
Example:
```
typedef unsigned long int Distance;
Distance i;
```

This code will declare a synonym for "unsigned long int". Now, synonym "Distance" can be used as type identifier to declare variable i of type "unsigned long int".

asm Declaration

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

Example:
```
asm {
   MOVLW 3
   MOVWF PORTB
}
```

This code will turn ON LED0 and LED1 on PORTB.

FUNCTIONS

Function Declaration

Syntax:
```
type function_name(parameter-declarator-list);
```
Example:
```
int add(int a, int b);
```

This will declare a function named `add` that accepts two parameters of type `int`.

Function Definition

Syntax:
```
type function_name(parameter-declarator-list){
   function body
}
```
Example:
```
int add(int a, int b){
   return a + b;
}
```

We can call it to calculate sum of two numbers:

```
int c;
c = add(4, 5);
```

Variable c will then be 9.

OPERATORS

mikroC recognizes following operators:

- Arithmetic Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Reference/Indirect Operators (see Pointers)
- Relational Operators
- Structure Member Selectors (see Structure Member Access)
- Comma Operator ,
- Conditional Operator ? :
- Array subscript operator [] (see Arrays)
- Function call operator () (see Function Calls)
- sizeof Operator
- Preprocessor Operators # and ## (see Preprocessor Operators)

Operators Precedence and Associativity

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

Arithmetic Operators

Operator	Operation	Precedence
+	addition	12
-	subtraction	12
*	multiplication	13
/	division	13
%	returns the remainder of integer division (cannot be used with floating points)	13
+ (unary)	unary plus does not affect the operand	14
- (unary)	unary minus changes the sign of operand	14
++	increment adds 1 to the value of the operand	14
--	decrement subtracts 1 from the value of the operand	14

Relational Operators

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

Note:
Use relational operators to test equality or inequality of expressions. All relational operators return true or false.

Bitwise Operators

Operator	Operation	Precedence
&	bitwise AND; returns 1 if both bits are 1, otherwise returns 0	9
|	bitwise (inclusive) OR; returns 1 if either or both bits are 1, otherwise returns 0	9
^	bitwise exclusive OR (XOR); returns 1 if the bits are complementary, otherwise 0	10
~	bitwise complement (unary); inverts each bit	10
<<	bitwise shift left; moves the bits to the left, see below	10
>>	bitwise shift right; moves the bits to the right, see below	10

Examples:
```
operand1  :            %0001 0010
operand2  :            %0101 0110
------------------------------
operator  &  :         %0001 0010
operator  |  :         %0101 0110
operator  ^  :         %0100 0100
```

Examples:
```
operand  :             %0101 0110
------------------------------
operator  ~  :         %1010 1001
operator  <<  :        %1010 1100
operator  >>  :        %0010 1011
```

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

Logical Operators

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

Operands of logical operations are considered `true` or `false`, that is non-zero or zero. Logical operators always return 1 or 0.

Example:
```
if (SW1 or SW2) LED1 = 1; else LED2 = 1;
```

If variable `SW1` or variable `SW2` is `true` (non-zero) then expression `(SW1 or SW2)` is equal 1 (`true`) and LED1 is turned ON. In case that both variables `SW1` and `SW2` are equal 0 (`false`) then `else` statement is executed and `LED2` is turned ON.

Conditional Operator ? :

Syntax:
```
expr1 ? expr2 : expr3
```

Example:
```
(i > 0) ? LED1 = 1 : LED2 = 1;
```
if variable `i` is greater then 0 `LED1` will be turned ON else `LED2` will be turned ON.

ASSIGNMENT OPERATORS

Simple Assignment Operator

Syntax:

```
expression1 = expression2
```

Example:

```
int  a;
a = 5;
```

This code declares variable a and assigns value 5 to it.

Compound Assignment Operators

Syntax:

```
expression1 op= expression2
```

where op can be one of binary operators +, -, *, /, %, &, |, ^, <<, or >>.
Compound assignment has the same effect as:

```
expression1 = expression1 op expression2
```

Example:

```
counter = counter + 1;
```

is the same as:

```
counter += 1;
```

Example:

```
items = items * 3;
```

is the same as:

```
items *= 3;
```

Sizeof Operator

Prefix unary operator sizeof returns an integer constant that gives the size in bytes of how much memory space is used by its operand.

Example:

```
sizeof(char)              /* returns 1 */
sizeof(int)               /* returns 2 */
sizeof(unsigned long)     /* returns 4 */
```

COMMA EXPRESSIONS

One of the specifics of C is that it allows you to use comma as a sequence operator to form the so-called comma expressions or sequences. It is formally treated as a single expression.

Syntax:

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

This results in the left-to-right evaluation of each expression, with the value and type of the last expression (expression_n) giving the result of the whole expression.

Example:

```
int i, j, array[5];
i = j = 0;
array[j+2, i+1] = 1;
```

This code declares variables i , j, and array of 5 integer elements. The last line of code is the same as if we wrote array[1] = 1; because the value of comma expression j+2, i+1 is value of i+1.

STATEMENTS

Statements can be roughly divided into:

- Labeled Statements
- Expression Statements
- Selection Statements
- Iteration Statements (Loops)
- Jump Statements
- Compound Statements (Blocks)

Labeled Statements

Syntax:

```
label_identifier : statement;
```

A statement can be labeled for two reasons:

1. The label identifier serves as a target for the unconditional goto statement,

Example:

```
loop : Display(message);
goto loop;
```

This is infinite loop that calls the Display function.

2. The label identifier serves as a target for the switch statement. For this purpose, only case and default labeled statements are used:

```
case constant-expression : statement
default : statement
```

For more information see switch statement.

SELECTION STATEMENTS

If Statement

Syntax:

```
if (expression) statement1 [else statement2]
```

Example:

```
if (movex == 1) x = x + 20; else y = y - 10;
```

Note:
The else keyword with an alternate statement is optional.

Switch Statement

Syntax:

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

Example:

```
switch (input) {
  case 1 : LED1 = 1;
  case 2 : LED2 = 1;
  case 3 : LED3 = 1;
  default : LED7 = 1;
}
```

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

ITERATION STATEMENTS

While Statement

Syntax:

```
while (expression) statement
```

Example:

```
int s, i;
s = i = 0;
while (i < 6) {
s = s + 2;
i = i + 1;
}
```

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

Do Statement

Syntax:

```
do statement while (expression);
```

Example:

```
int s, i;
s = i = 0;
do {
s = s + 2;
i = i + 1;
} while (i < 7);
```

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

For Statement

Syntax:

```
for ([init-exp]; [condition-exp]; [increment-exp]) statement
```

Example:

```
for (s = 0, i = 0; i < 5; i++) {
s += 2;
}
```

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

JUMP STATEMENTS

Break Statement

Use the break statement within loops to pass control to the first statement following the innermost `switch`, `for`, `while`, or `do` block.

Example:

```
int i = 0, s = 1;   // declares and initiate variables i and s
while (1) {   // infinite loop
   if (i == 4) break;
   s = s * 2;
   i++;
}
```

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

Continue Statement

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

Example:

```
int i = 0, s = 1;   // declares and initiate variables i and s
while (1) {   // infinite loop
   s = s * 2;
   i++;
   if (i != 4) continue;
   break;
}
```

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

Goto Statement

Syntax:

```
goto label_identifier;
```

Example:

```
loop : Display(message);
goto loop;
```

This is infinite loop that calls the Display function.

Return Statement

Use the `return` statement to exit from the current function back to the calling routine, optionally returning a value.

Syntax:

```
return [expression];
```

Example:

```
...
c = add(4, 5);
...
int add(int a, int b) {
return a + b;
}
```

Compound Statements (Blocks)

A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces `{}`.

PREPROCESSOR

Preprocessor Directives

mikroC supports standard preprocessor directives:

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

MACROS

Defining Macros

Syntax:

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

Example:

```
#define ERR_MSG "Out of range!"
...
main() {
  ...
  if (error) Show(ERR_MSG);
  ...
}
```

Compiler will replace ERR_MSG with string “Out of range!” and when Show function is executed it will display “Out of range!”.

Macros with Parameters

Syntax:

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

Example:

A simple macro which returns greater of its 2 arguments:

```
#define MAX(A, B) ((A) > (B)) ? (A) : (B)
...
x = MAX(a + b, c + d);
```

Preprocessor will transform the previous line into:

```
x = ((a + b) > (c + d)) ? (a + b) : (c + d)
```

Undefining Macros

Syntax:

```
#undef macro_identifier
```

Directive `#undef` detaches any previous token sequence from the *`macro_identifier`*; the macro definition has been forgotten, and the *`macro_identifier`* is undefined.

Note:

You can use the #ifdef *and* #ifndef *conditional directives to test whether any identifier is currently defined or not.*

File Inclusion

Syntax:

```
#include <header_name>
#include "header_name"
```

Explicit Path

Example:

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

CONDITIONAL COMPILATION

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

Syntax:

```
#if constant_expression_1
<section_1>

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

[#else
<final_section>]

#endif
```

Example:

```
#if OSC == 8
  ...
// code for oscillator 8Hz
  ...
#elif OSC == 10
  ...
// code for oscillator 10Hz
  ...
#else
...
// code for other oscillators
...
#endif
```

In this example only one code section is compiled regarding of oscillator frequency.

Directives #ifdef and #ifndef

Syntax:

```
#ifdef identifier   // or
#ifndef identifier
```

Example:

```
#ifndef MODULE
...
// code that will be compiled
// if identifier MODULE is not
// defined whith #define
// directive
...
#endif
```