This reference guide is intended to quickly introduce user’s to C language syntax with the aim to easily start programming microcontrollers 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
The following table shows the available escape sequences in mikroC:

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

**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.
mikroC Quick Reference Guide

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

**KEYWORDS**

```c
asm enum signed
auto extern sizeof
break float static
case for struct
cchar goto switch
cconst if typedef
ccontinue int union
default long unsigned
do register void
double return volatile
derule
```

**FUNDAMENTAL TYPES**

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned) char</td>
<td>8-bit</td>
<td>0 .. 255</td>
</tr>
<tr>
<td>signed char</td>
<td>8-bit</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>(signed) short (int)</td>
<td>8-bit</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>unsigned short (int)</td>
<td>8-bit</td>
<td>0 .. 255</td>
</tr>
<tr>
<td>(signed) int</td>
<td>16-bit</td>
<td>-32768 .. 32767</td>
</tr>
<tr>
<td>unsigned (int)</td>
<td>16-bit</td>
<td>0 .. 65535</td>
</tr>
<tr>
<td>(signed) long (int)</td>
<td>32-bit</td>
<td>-2147483648 .. 2147483647</td>
</tr>
<tr>
<td>unsigned long (int)</td>
<td>32-bit</td>
<td>0 .. 4294967295</td>
</tr>
<tr>
<td>float</td>
<td>32-bit</td>
<td>±1.17549435082E-38 ..  ±6.8054774407E38</td>
</tr>
<tr>
<td>double</td>
<td>32-bit</td>
<td>±1.17549435082E-38 ..  ±6.8054774407E38</td>
</tr>
<tr>
<td>long double</td>
<td>32-bit</td>
<td>±1.17549435082E-38 ..  ±6.8054774407E38</td>
</tr>
</tbody>
</table>

**Enumeration**

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

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

**DERIVED TYPES**

**ARRAYS**

**Array Declaration**

Syntax:
```c
type array_name[constant-expression];
```

Example:
```c
int array_one[7]; /* an array of 7 integers */
```

**Array Initialization**

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

Note:
You must initialize pointers before using them!
Now pointer p points to variable a.
*p = 5; assigns value 5 to variable a.

Multi-dimensional Arrays
Example:
```c
float m[50][20]; /* 2-dimensional array of size 50x20 */
```

STRUCTURES

Structure Declaration and Initialization
Syntax:
```c
struct tag { member-declarator-list };
```
Example:
```c
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:
```c
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
```

UNIONS

Union Declaration
Syntax:
```c
union tag { member-declarator-list };
```

Union Member Access
Example:
```c
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!
```

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.

BIT FIELDS

Bit Fields Declaration
Syntax:
```c
struct tag { bitfield-declarator-list };
```
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>
<thead>
<tr>
<th>Precedence</th>
<th>Operands</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>( ) [ ] . -&gt;</td>
<td>left-to-right</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>! ~ ++ -- + * &amp;</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>&lt;&lt; &gt;&gt;</td>
<td>left-to-right</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left-to-right</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>== != &amp; = ^</td>
<td>left-to-right</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>&amp;</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></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>&amp;&amp;</td>
<td>left-to-right</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
<td></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>*= /= %= += -= &amp;= ^=</td>
<td>= &lt;&lt;= &gt;&gt;=</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>,</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>

## Arithmetic Operators

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

## Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal</td>
<td>9</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>9</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>10</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>10</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>10</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>10</td>
</tr>
</tbody>
</table>
Note:
Use relational operators to test equality or inequality of expressions. All relational operators return true or false.

Bitwise Operators

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

Examples:

<p>| operand1 : | %0001 0010 |</p>
<table>
<thead>
<tr>
<th>operand2 :</th>
<th>%0101 0110</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator &amp; :</td>
<td>%0001 0010</td>
</tr>
<tr>
<td>operator</td>
<td>/ : %0101 0110</td>
</tr>
<tr>
<td>operator ^ :</td>
<td>%0100 0100</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>14</td>
</tr>
</tbody>
</table>

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

Example:

```c
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.

Example:

```c
(i > 0) ? LED1 = 1 : LED2 = 1;
```
If variable i is greater then 0 LED1 will be turned ON else LED2 will be turned ON.
Statements can be roughly divided into:

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

**Simple Assignment Operator**
Syntax: 
\[ \text{expression}1 \ = \ \text{expression}2 \]
Example:
\[
\begin{align*}
\text{int} & \ a; \\
& \ a \ = \ 5;
\end{align*}
\]
This code declares variable \( a \) and assigns value 5 to it.

**Compound Assignment Operators**
Syntax:
\[ \text{expression}1 \ \text{op} = \ \text{expression}2 \]
where \( \text{op} \) can be one of binary operators +, -, *, /, %, |, ^, <<, or >>.
Compound assignment has the same effect as:
\[ \text{expression}1 \ = \ \text{expression}1 \ \text{op} \ \text{expression}2 \]

Example:
\[
\begin{align*}
\text{counter} & \ = \ \text{counter} \ + \ 1; \\
\text{is the same as:} & \ \\
\text{counter} & \ += \ 1;
\end{align*}
\]

**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:
\[
\begin{align*}
\text{sizeof} & (\text{char}) \ /* \text{returns 1} */ \\
\text{sizeof} & (\text{int}) \ /* \text{returns 2} */ \\
\text{sizeof} & (\text{unsigned long}) \ /* \text{returns 4} */
\end{align*}
\]

**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:
\[ \text{expression}_1, \ \text{expression}_2, \ldots \text{expression}_n; \]
This results in the left-to-right evaluation of each expression, with the value and type of the last expression (\( \text{expression}_n \)) giving the result of the whole expression.
Example:
\[
\begin{align*}
\text{int} & \ i, \ j, \ \text{array}[5]; \\
i & \ = \ j \ = \ 0; \\
\text{array}[j+2, \ i+1] & \ = \ 1;
\end{align*}
\]

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

**If Statement**

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

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

**Switch Statement**

Syntax:
```
switch (expression) {
    case const-expression_1 : statement_1;
    case const-expression_2 : statement_2;
    ...
    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 different then ones mentioned in value list in case statement then default statement is executed.

**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.
**Jump Statements**

**For Statement**
Syntax:
```c
for ([init-exp]; [condition-exp]; [increment-exp]) statement
```
Example:
```c
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.

**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:
```c
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:
```c
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:
```c
goto label_identifier;
```
Example:
```c
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:
```c
return [expression];
```
Example:
```c
... 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"
```
### 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
```

Page 11
---