

# MicroTech

## TinyC Programming Reference

Version 4.1

[www.mcu.hk](http://www.mcu.hk)

*It is assumed that the reader has a basic knowledge of the C programming language. This document is not intended to provide a detailed explanation of C or how to use it – that is left to the many text books written on the subject. However some description of the basic architecture and structure of C is necessary to help users start programming in TinyC and also to explain some of the differences between TinyC and standard ANSI C.*

July 2011

## Introduction

**TinyC** is a C-like programming language used to develop programs for **MicroTech MT chips**. Although tiny in terms of data type and language constructs, and with just over a dozen functions, it is still powerful enough for developing complex programs. It is easy to learn and can be mastered in a short period of time. A TinyC program can have a maximum size of 32K program words (64K bytes).

The syntax of the TinyC language follows the standard ANSI C language. It has fewer language constructs but it does support up to five I/O pin interrupts, one timer interrupt and recursion. One of the major features of MT chips is that all the I/O pins can be used for both input and output, generating pulses for speakers/devices, moving RC servos plus any five I/O pins can be defined as external interrupt inputs.

TinyC was designed and developed by Victor Lee of MicroTech Systems Limited ([www.mcu.hk](http://www.mcu.hk)) with contributions from Frank Crivelli of Ozitronics ([www.ozitronics.com](http://www.ozitronics.com)).

## What are MT chips?

MT chips are based on the latest generation of 8051, Atmel AVR or ARM core flash microcontrollers that have been programmed with the MT system code. The MT system code is a 16-bit run-time interpreter that executes programs written in the TinyC language. The MT system code has a built-in downloader that enables the chip to be reprogrammed (>100,000 times) directly via a serial connection to a PC, eliminating the need for an expensive IC programmer. This makes developing TinyC programs for MT chips a simple and easy task.

For further details of MT chips and how to use them please refer to the '**MT Chips Hardware Reference**'.

## Differences between TinyC and standard ANSI C

TinyC was not designed to be ANSI C compatible. It is a 'tailor-made' version aimed at developing programs for microcontrollers. Users should be fully aware of TinyC's limitations (or added new features) before writing any programs.

The following details the main differences between TinyC and standard ANSI C:

- No need for function parameter declarations.
- Only one variable type – *int* (integer). No support for floating point numbers.
- Both global and local variables are initialized to 0 at program startup.
- No need to specify function return type (if used) – always assumed to be *int*.
- TinyC does not support **string** type, but supports literal strings and string constants.
- Starting in version 4.1, TinyC supports both parameter passing by name (value) and by reference (address). Previous versions support parameter passing by name only.
- Extra iteration construct – **loop**, in addition to *while*, *do..while* and *for loops*.
- In TinyC, the <expression> parts of looping statements cannot be empty.
- The <expression> in the **case** part within a **switch-case** statement can be any expression, not necessary a constant label expression.
- TinyC has numbers of built-in functions to access connected hardware easily, such as, SONY PS2 Controller, 24C02 & 93C56 memory ICs, R/C servos, IR sensor, etc...

## Sample Programs

A number of TinyC sample programs are included with the MicroTech software pack. You are encouraged to study these programs and experiment with them. This will make it easier to learn the language and speed up your program development time.

**TinyC Language Limits**

Maximum <b>code size</b> (in 16-bits words)	32760
Maximum number of characters in a single <b>input line</b>	300
Maximum number of characters in an <b>identifier</b>	30
Maximum number of characters in a <b>file name</b>	100
Maximum number of <b>break</b> statements within a looping statement	100
Maximum number of <b>continue</b> statements within a looping statement	100
Maximum number of <b>functions</b> in the whole program	100
Maximum number of <b>function calls</b> within a function	100
Maximum number of <b>timer interrupts</b> in the whole program	1
Maximum number of <b>I/O pin interrupts</b> in the whole program	5
Maximum number of <b>string constants</b> in the whole program	200
Maximum number of <b>numeric constants</b> in the whole program	200
Maximum number of <b>global variables</b> in the whole program	200
Maximum number of <b>local variables</b> within a function	200

## VARIABLES

A variable is used to hold data in a program. A variable represents a location in the MT chip's memory. Data can be put into this location and retrieved from it. Every variable has two parts - a name and a data type.

Syntax: `int <name> {,<name>};`

### Variable Names

Valid names can consist of letters, numbers and the underscore, but may not start with a number. A variable name may not be a TinyC keyword ie. if, else, loop or function ie. pinhigh, readm, etc.

Variable names are case sensitive. So **cat**, **CAT** and **Cat** are all names for **different** variables. This is not a recommended programming practice as it can lead to confusion and errors.

### Variable Data Types

There is only **one** data type in TinyC – **int** (integer), which is 16 bits. Therefore the range of available numbers are –32768 to 32767 for signed integers and 0 to 65535 for unsigned integers.

Integer variables are also used to represent character and Boolean data types. During any Boolean expression evaluation the number '0' is interpreted as FALSE and any other non-zero numbers (including negative numbers) are TRUE. Character variables only use the least significant 8 bits – the rest are ignored.

### Variable Declaration

A declaration is used to specify the name and type of a variable to the compiler. Variables **must** be declared before they can be used in a program. Variables can be declared as **local** or **global**.

**Local variables** are variables declared inside a function. They only exist within the function in which they are declared and can only be accessed within that function. They are said to be 'local to that function' and cease to exist once the function terminates. This applies to the main() function as well. Any variables declared in **main()** are not available to any functions called within main(). Parameter passing (described later) is used to pass local variables to other functions.

**Global variables** are variables declared outside any function, including main(). They are called 'global' because exist outside any function and therefore can be accessed by ALL functions within the program. There is no need to pass global variables to functions. Take care when using global variables because they can be changed by any function in your program.

### Variable Initialisation

**All** variables in TinyC are assigned a value of '0' (zero) when declared. So, if a variable requires an initial value of zero it is already done. For other values an assignment statement or initializer will be required, i.e. `a = 5` or `int a = 5`.

Variables can be assigned a value in **decimal** or **hexadecimal**. Hexadecimal numbers begin with '0x'.

```
int a, myData;           // these are global variables - declared outside main()

main()
{
    int x = 12345; // declared a variable with initialized value
    int b;        // this is a local variable available only in main()
    a = 10;       // assigns the decimal value 10 to global variable 'a'
    b = 0x10;     // assigns the hexadecimal value 10 to 'b' (0x10 = 16 in decimal)

    ..... (more code)
}

anotherfunction()
{
    int b;        // this variable 'b' is not the same as the 'b' declared in main()
                  // even though it has the same name. It is local to this function
                  // it ceases to exist once this function terminates

    ..... (more code)
}
```

## OPERATORS

An operator is used to combine terms in an expression and produce a result eg.  $a = 1 + 2$ . Operators can be broken down into the following groups:

Operator	Example	Description
<b>Arithmetic Operators</b> (the result will be truncated to fit into 16 bits)		
+	$a + b$	a plus b                      addition
-	$a - b$	a minus b                    subtraction
*	$a * b$	a times b                    multiplication
/	$a / b$	a divided by b              division
%	$a \% b$	remainder of $a / b$ modulo division
++	$a++$	post-increment            a is incremented <b>after</b> it is used
++	$++a$	pre-increment             a is incremented <b>before</b> it is used
--	$a--$	post-decrement            a is decremented <b>after</b> it is used
--	$--a$	pre-decrement             a is decremented <b>before</b> it is used
<b>Bitwise Operators</b>		
&	$a \& b$	binary AND
	$a   b$	binary INCLUSIVE OR
^	$a \wedge b$	binary EXCLUSIVE OR (XOR)
~	$\sim a$	1's complement of a      each bit in a is 'flipped' ( $1 \rightarrow 0, 0 \rightarrow 1$ )
>>	$a \gg b$	a shifted right by b bits   each bit in a is shifted right by b positions
<<	$a \ll b$	a shifted left by b bits    each bit in a is shifted left by b positions
<b>Relational Operators</b>		
==	$a == b$	is a equal to b?            Yes – returns 1, No – returns 0
!=	$a != b$	is a <b>not</b> equal to b?        Yes – returns 1, No – returns 0
>	$a > b$	is a greater than b?        Yes – returns 1, No – returns 0
<	$a < b$	is a less than b?            Yes – returns 1, No – returns 0
>=	$a \geq b$	is a greater than or equal to b?    Yes – returns 1, No – returns 0
<=	$a \leq b$	is a less than or equal to b?    Yes – returns 1, No – returns 0
<b>Logical Operators</b>		
&&	$a \&\& b$	logical AND                 Result = 1 if both a and b are NOT zero, 0 otherwise
	$a    b$	logical OR                  Result = 1 if either a or b is NOT zero, 0 otherwise
!	$!a$	logical NOT                 Result = 1 if a is zero, 0 otherwise
<b>Assignment Operators</b>		
=	$a = b$	a equals b
+=	$a += b$	$a = a + b$
-=	$a -= b$	$a = a - b$
*=	$a *= b$	$a = a * b$
/=	$a /= b$	$a = a / b$
%=	$a \% = b$	$a = a \% b$
&=	$a \& = b$	$a = a \& b$
=	$a   = b$	$a = a   b$
^=	$a \wedge = b$	$a = a \wedge b$
>>=	$a \gg = b$	$a = a \gg b$
<<=	$a \ll = b$	$a = a \ll b$
<b>Others</b>		
?:	ternary	$x?y:z$ , if x is true then y else z (applicable to expressions only)
()	func()	function call
[]	$a[9]$	array reference $a[8]$ refers to 9 <sup>th</sup> element of the array (0 = 1 <sup>st</sup> element)
&	$\&a$	address of a
*	$*a$	contents of the variable at address pointed to by a

## Increment/Decrement Operators

The traditional method of incrementing numbers is  $a = a + 1$ . In C, you can also use the `++` operator to perform the same function. So, `a++` also adds 1 to the value of `a`. Similarly you can also decrement a variable by using `a--`.

In the case of post-increment/decrement, the variable is used first before it is incremented or decremented, whereas in case of pre-increment/decrement, the operation is applied before the value of variable is taken.

```
int a, b, c;

a = 1;           // assign 'a' = 1
b = a++;        // 'b' is assigned the value of 'a' before 'a' is incremented (b = 1)
c = a;          // 'c' = 2 because 'a' was incremented in the previous line
```

## Bitwise Operators

The **AND** operator `&` copies a 1 to the result if that bit is 1 in **both** variables.

```
int a, b, c;

a = 60;          // 60 = 0011 1100 in binary
b = 13;          // 13 = 0000 1101 in binary
c = a & b;       // 12 = 0000 1100 in binary
```

The **OR** operator `|` copies a bit to the result if that bit is 1 in **either** variable.

```
int a, b, c;

a = 60;          // 60 = 0011 1100
b = 13;          // 13 = 0000 1101
c = a | b;       // 61 = 0011 1101
```

The **XOR** operator `^` copies a 1 to the result if that bit is set in **one** of the variables (but not both)

```
int a, b, c;

a = 60;          // 60 = 0011 1100
b = 13;          // 13 = 0000 1101
c = a ^ b;       // 49 = 0011 0001
```

The **1's complement** operator `~` 'flips' each bit in the variable

```
int a;

a = 6;           // 6 = 0000 0110
a = ~a;         // 249 = 1111 1001
```

The **bit shift operators** `>>` and `<<` move the bits in a variable right or left by a number of positions. Binary 0s are shifted into the variables to fill the 'vacant' bit positions.

```
int a;

a = 11;          // 11 = 0000 1011
a = a << 4;      // 'a' = 1011 0000 ('a' shifted left by 4 bit positions)
a = a >> 2;      // 'a' = 0010 1100 ('a' shifted right by 2 bit positions)
```

## Logical Operators

The **logical AND** operator `&&` returns true (1) if **both** sides of the expression is NOT 0.

The **logical OR** operator `||` returns true (1) if **one** side of the expression is NOT 0.

The **logical NOT** operator `!` returns true (1) if the variable IS 0.

```
a = 3;
b = 2;
c = 0;

return (a && b); // returns true (1)
return (a && c); // returns false (0) because both sides are NOT 0

return (c || a); // returns true (1) because one side is NOT 0
```

```
return (!a);           // returns false (0) because 'a' is NOT 0
return (!c);          // returns true (1) because 'c' IS 0
```

## Assignment Operators

The assignment operator '=' is obvious – it takes the variable or expression on the right and assigns it to the variable on the left. The other assignment operators, such as '+=', '/=', etc, are a shorthand way of writing the expression.

```
a = 1;                // assigns the value '1' to 'a'
a = a + 5;            // adds '5' to the current value of 'a'

a += 5;               // another way of writing the previous expression - same result
```

## Operator Precedence in C

The following table shows the order in which multiple operators in an expression are evaluated. Operators on top have a higher priority and are evaluated first.

Confusing the evaluation order of multiple operators in an expression is a common programming error. The best solution is the use brackets '(' )' to set the order. When in doubt – use brackets!

( ) [ ]	functions, arrays
! ~ ++ --	logical NOT, 1's complement, increment/decrement
* / %	multiplication, division, modulo division
+ -	addition, subtraction
<< >>	shift left, shift right
< <= > >=	relational
== !=	equals, not equals
&	bitwise AND
^	bitwise XOR
	bitwise OR
&&	logical AND
	logical OR
= += -= *= /= %= >>= <<= &= ^=  = ?:	Assignment and ternary operator

```
a = 2 + 3 * 5 ;      // 'a' = 17 because '**' has precedence over '+'
b = (a + 1) / 3;     // 'b' = 6 because the use of brackets gives '+' precedence over '/'
```

## POINTERS

**&variable** // returns the address of 'variable'  
**\*pin ter** // returns the variable at the address 'pointer'

The 'address of' operator, **&**, returns the address of a variable.  
 The 'pointer' operator, **\***, holds the address of a variable and is used to reference the variable indirectly.

```
main()
{
    int a;           // integer variable declaration
    int *ptr;        // int pointer declaration

    a = 5;           // 'a' equals 5
    ptr = &a;        // 'ptr' now contains the address of 'a'

    *ptr += 1;      // adds 1 to variable referred to by 'ptr' (a = 6)
}
```

## ARRAYS

An array is a group of variables of the same data type stored sequentially in memory. All the elements of an array must be of the same data type (in TinyC there is only one data type – **int**). The size of an array (number of elements) **must** be specified when declared.

Elements of an array are referenced by an index number in square brackets '[]' following the array name. Array index numbers start at zero.

```
int num[3];           // declares an array of 3 integers called 'num'
                    // the individual elements are referenced as 'num[0], num[1], num[2]'
```

TinyC only supports single dimensional array and does not support string arrays.

Syntax: **int <name>[ <constant expression> ] {,<name>[<constant expression> ]};**

Arrays, like any other variable, can be assigned with values when declared. This method is called array initialization. In TinyC, the number of values to assign must matches the array size.

```
Int   num[3] = { 10, 20, 30 };
```

## FUNCTIONS AND RETURN TYPE

A function is a self-contained block of code that performs some task. Functions are called from other functions. They can be passed parameters and can return a value. Functions are often used to divide the program into smaller sections to make it easier to understand and debug. It also allows common tasks to be grouped together and called from different parts of a program.

Syntax: **func(<parameter>,...)**  
**{..}**

A C program can have many functions but it must **always** have a **main()** function. The **main()** function is the entry point to the whole program. In TinyC, function parameter declarations are not required and the return variable, if used, is always of type **int**.

Functions that return values cannot appear as standalone statements and must be part of an expression or on the right-hand side of an assignment statement.

```
if ((multiply(2,3) > 10) break; // using the return value of a function in an expression
a = multiply(2,3);             // using the return value of a function in an assignment statement
```

## PARAMETERS AND PARAMETER PASSING

A parameter is a piece of data that is passed to another function. The data can be a **constant** or a **variable**.

```
multiply(2, 3);           // passing 2 constants as parameters to the 'multiply' function
multiply(a, b);          // passing 2 variables as parameters
```

In ANSI C there are two ways of passing parameters to functions – by value or by reference.

**Passing by value** is also known as **call by value**. A copy of the variable is passed to the function, not the variable itself. If the function changes the copy the original remains unaltered.

**Passing by reference** is also known as **call by reference**. The function is not passed a copy of the data but a pointer to the data. If the function changes the data the original is altered. Note that arrays can only be passed by reference.

## PROGRAM FLOW CONTROL

The keywords **return**, **break** and **continue** are used to control program flow, as follows:

A **return** statement terminates a function, optionally returning a value. Execution continues in the calling function.

A **break** statement is used to exit a loop statement. Execution continues at the statement following the loop statement. With nested loops, a break statement terminates the enclosing loop ie. the loop it is used in. The outer loop is not affected.

A **continue** statement skips the rest of statements up to end of its innermost enclosing loop. Execution continues at the end of the loop statement. This is different to break which terminates the loop.

```
main()                // program starts here
{
  int a, b;           // declare local variables – both default to '0'

  loop {
    b = double(a);    // function call with parameter passing
                    // the called function returns a value which is assigned to 'b'
    if (a++ == 5) continue; // if a equals 5 then skip the rest (returns to top of loop)
                    // Note: 'a' is used first before it is incremented
    writei(232, b, 0); // output the result of 'double a' to serial port
    writec(232, ' ', 0); // output a space to serial port
    if (a > 10) break;
  }
}

double(a)
{
  return a*2;        // returns 2 times the value of the variable passed to it
}
```

The output of this program is “0 1 4 6 8 12 14 16 18 20”.

Note that 10 (2 x 5) is skipped – the result of the expression containing **continue**.

## CONDITIONAL PROCESSING

Syntax: **if (<expr>) <statement1>; [else <statement2>; ]**

Syntax: **switch (<expr>) { [case <expr>: <statement>; break;] [default: <statement>; break;] }**

The **if** statement is used to execute a block of code based on whether a **test condition** is true. If the condition is true the block of code is executed, otherwise it is skipped. The **else** statement is optional and provides a way to execute one block of code if a condition is true, another if it is false.

```
main()
{
  int number;

  number = 10;
  if (number == 5) writes("$Number equals 5",0); // if without else statement
  if (number > 5) writes("$Number is greater than 5",0); // if..else statement
  else writes("$Number is less than or equals to 5",0);
}
```

The **if...else** statement can also be written using the **ternary** operator **'?:'**. In some cases the use of the ternary operator improves the legibility of the program. The ternary operator applies to expressions only!

For example, the statement

```
if(x > 10) y = 100; else y = 200;
```

could be re-written as follows:

```
y = (x > 10) ? 100 : 200;
```

The **switch** statement can be viewed as a series of **if..else..if..else** constructs which is very useful for testing a number of expressions.

#### Example: run-rc.prg

```
// Control robot movements by an IR remote controller
#include "motor.h"
main()
{
  writes("rc", 500);
  loop switch(ircode()) {
    case 2: drive( 1,  1, 0); break;           // forward
    case 8: drive(-1, -1, 0); break;         // backward
    case 1: drive(-1,  1, 0); break;         // spin left
    case 3: drive( 1, -1, 0); break;         // spin right
    case 4: drive(-1,  1, 200); drive(1, 1, 0); break; // turn left;
    case 6: drive( 1, -1, 200); drive(1, 1, 0); break; // turn right
    case 5: stop all(); break;               // standstill
    case 0: stop all(); writes("end", 2000); exit(); break; // stop and exit
  }
}
```

## LOOPING

Loops are used to execute a block of code a number of times or even continuously. In either case a condition is tested, within the loop, to allow the loop to terminate. The condition can be either updated within the loop itself or by an external event such as an input going high or low. Without this test the loop would never terminate, creating a programming bug known as an 'infinite loop'.

Standard ANSI C has a number of looping statements – **while**, **do..while** and **for..next**. TinyC supports all of them, and has an additional loop statement – **loop**! It is easy to construct the equivalent standard C loop statements by using the **loop** statement alone. Note that **loop** can be nested ie. loops within loops.

Syntax: **loop { <statement>; }**  
**while(<expression>) { <statement>; }**  
**do { <statement>; } while(<expression>;)**  
**for(<expression>; <expression>; <expression>) { <statement>; }**

**Note:** In TinyC, the <expression> parts cannot be empty.

Examples:

Standard ANSI C 'loop' statements	TinyC equivalent
<pre>while (a &lt;= 10) {   b = b + a++; }</pre>	<pre>loop {   if (a &gt; 10) break;   b = b + a++; }</pre>
<pre>do {   b = b + a++; } while (a &lt;= 10);</pre>	<pre>loop {   b = b + a++;   if (a &gt; 10) break; }</pre>
<pre>for (a = 1; a &lt;= 10; a++) {   b = b + a; }</pre>	<pre>a = 1; loop {   if (a &gt; 10) break;   b = b + a++; }</pre>
<pre>while(a);</pre>	<pre>loop { if (!a) break; }</pre>

Example: loops.prg

```
// Test all the looping statements

#define WAIT 500
#define DLY 300

main()
{
    int i;

    loop { // test repeatedly

        writes("while", WAIT);
        i = 1;
        while(i <= 7) {
            writei('d', 1000+i, DLY);
            i++;
        }
        delay(WAIT);

        writes("do-while", WAIT);
        i = 1;
        do {
            writei('d', 2000+i, DLY);
            i++;
        } while(i <= 10);
        delay(WAIT);

        writes("for", WAIT);
        i = 0;
        for(i=5; i <= 100; i+=10) {
            writei('d', 3000+i, DLY);
        }
        delay(WAIT);
    }
}
```

## COMMENTS

***/\* comment \*/***

Classic C comment. It starts with a */\** and ends with a *\*/*. Nested comments are allowed, but must be in pairs.

***// comment***

One line C++ comment. Simply starts with *//*.

```
main( )
{
    int a;      /* declare an integer variable */

    a = 5;     // initialise variable
}
```

## COMPILER DIRECTIVES

Compiler directives are executed by the TinyC compiler before the program is compiled. These controlling constructs are very useful in controlling which parts of the program source is submitting to compilation, and they can easily helps to maintain the program for later modification by using labels for frequently used constants. Please be aware that **nested directives** are not supported In TinyC.

***{\$Prog N}***

***{\$Run N}***

**Note:** N is a number between 0 and 4 (i.e. program storage location).

The above 2 directives are applicable to MT chips with multiple program storage locations. Instead of setting the I/O pins manually, the MT Downloader stores the program into the **N** storage location (set by **Prog**) and

set the default program to run (set by **Run**) after resetting the MCU. However, manual hardware settings will override these settings. These directives have no effect on MT chips with a single storage or MicroBoards.

### **#include "file.h"**

Includes the file "file.h" into the program. Nested includes are not allowed. Include files are are **not** precompiled lib files as in other C compilers. They are simply TinyC format source code files. Using include files allows a large program to be broken up into smaller files which makes them easier to edit and debug. They are also used to group all program definitions together in a single file.

### **#define MAX 0xFF // equals to 255 in decimal**

Assigns the hexadecimal constant '0xFF' to 'MAX'. Wherever 'MAX' is used in the program it is replaced by '0xFF'.

### **#define SIZE 10\*5 // equals 50**

Assigns the decimal constant '50' to 'SIZE'. Wherever 'SIZE' is used in the program it is replaced by '50'.

### **#define CH 'A' // equivalent to #define CH 65, whereas 65 is 'A' ASCII code**

Assigns the character constant 'A' to 'CH'. Wherever 'CH' is used in the program it is replaced by 'A'.

### **#define MSG "Hello World"**

Assigns the string constant "Hello World" to label MSG, later be used in *writes()* function.

### **#define TinyC // equivalent to #define TinyC 1**

Assigns the decimal constant '1' to 'TinyC', use this method to control conditional compilation of program source codes.

Assigning constants to labels (or called macros) allows the program easier to modify in the future if required. Just one line needs to be edited rather than each line that would contain the constant. Using meaningful label names, this programming method also improves the readability of the program

```
{ $Prog 1 } // stores program in storage location 1
{ $Run 1 } // set the default running program

#include "ascii.h" // include the file "ascii.h" in this program
// it contains the definitions for SPACE and BELL

#define MSG "Hello World" // a string constant
#define CH 'A' // a character constant
#define COMM 232 // a number constant
#define NUM 5 // another number constant
#define HEX 0xAA // a Hex number constant, equals to 10101010 in binary
#define SIZE (NUM * 10) // a bracketed constant expression

main( )
{
    writes(MSG, 1000); // displays the message
    writec(COMM, SPACE, 0); // output the space character to serial port
    writec(COMM, BELL, 0); // output the bell character to serial port
}

// here is the 'ascii.h' file
// it must be included in the program above otherwise there will be compile errors.

#define SPACE 0x20 // assign the label 'SPACE' the value of '0x20'
#define BELL 8 // assign the label 'BELL' the value of '8'
```

### **#undef <label>**

Undefines a previously defined label.

### **defined(<label>)**

This is also called the macro function is used to test whether a label name has been defined.

**#if** <constant-expression>

Conditional compilation of text, which depends on the result of the constant expression.

**#ifdef** <constant-expression>

Conditional compilation of text, which depends on whether or not a macro name has previously been defined.

**#ifndef** <constant-expression>

Opposite to **#ifdef**. The text is only compiled if the specified macro name has not previously been defined.

**#elif** <constant-expression>

Equivalent to **#else #if** combination.

**#else**

Associated with **#if**, **#ifdef** or **#ifndef** directives which provides an alternative text for compilation.

**#endif**

Associated with **#if**, **#ifdef** or **#ifndef** directives which provides a closing end.

**Note:** **#if**, **#ifdef**, **#ifndef**, **#elif**, **#else** and **#endif** directives **cannot** be nested.

```
#define M1 "Hello World"
#define M2 "This is great!"
#define COM 232
#define CH 'A'
#define A1
#define A2

//#undef A1           // test #undef

#if defined(A1)       // test #if & defined()
#define TEST 7654
int firstvar = 12345; // global variable
#else
#define TEST 3210
int firstvar = 54321; // global variable
#endif

//#ifndef A1          // test #ifndef
#ifdef A1             // test #ifdef
#define VAL 100
xxx(){int a=111; int b=222; writei('d', a, 1000); writei('d', b, 1000); }
#elif A2             // test #elif
#define VAL 200
xxx(){int a=333; int b=444; writei('d', a, 1000); writei('d', b, 1000); }
#else                // test #else
#define VAL 300
xxx(){int a=555; int b=666; writei('d', a, 1000); writei('d', b, 1000); }
#endif               // test #endif
```

## Summary of TinyC Functions

**Note:** Not all the functions are available on all types of MT chips.

### Output functions

pinhigh(), pinlow(), toggle(), writei(), writec(), writeb(), writes(), writem(), sound(), servo(), putchar(), putint(), puts()

### Input functions

pinstate(), readm(), readadc(), readpwm(), readsys(), ircode(), scankey(), getkey(), getchar()

### A/D conversion

readadc()

### PWM measurement

readpwm()

### D/A conversion

By using sys(50, x) (only on ARM-based MT chips)

### Sound generation

sound()

### Serial communication

setcomm(), putint(), putchar(), writes(), getchar()

### Data memory (24C02 and 93C56) management

readm(), writem()

### Time delay function

delay()

### RC servo control function

servo()

### Infra-Red decoder function

ircode()

### Keyboard input functions

scankey(), getkey() (MT-40r1 and MT-40r2 only)

### Byte output function

writeb()

### System management functions

sys(), readsys(), exit(), freemem(), random()

### SONY PS2 Controller

ps2(), ps2key(), ps2x(), ps2y(), ps2h(), ps2v()

**Description of TinyC functions**

Please refer to the sample programs supplied with the MT system software for examples on how these various functions are used.

**NOTE:**

1. Some functions, mainly the 'write' functions, behave differently between the Microboard and MT chips. The Microboard has a built in 7-segment display and the MT chips do not.
2. Some functions are hardware dependent and will only operate correctly on selected MT chips. Any unsupported functions are simply ignored.

**pinhigh(P)**

The pinhigh() function sets I/O pin 'P' high (1).

```
loop {
  pinhigh(0); delay(200);
  pinlow(0);  delay(200);
  toggle(1);  delay(200);
}
```

**pinlow(P)**

The pinlow() function sets I/O pin 'P' low (0).

**toggle(P)**

The toggle() function reverses the state of I/O pin 'P' ie. from 1 to 0 or 0 to 1

**pinstate(P)**

The pinstate() function returns the current state (1 or 0) of I/O pin 'P'.

**Example: timer.prg**

```
int x;

main()
{
  loop {
    if(!pinstate(14)) sys(1, 0); // Stop timer!
    if(!pinstate(13)) sys(1, 10); // reset timer, goes faster
    if(!pinstate(12)) sys(1, 500); // reset timer, goes slower
    if(!pinstate(11)) sys(1, 100); // back to original setting
  };
}

test timer() timer 100
{
  writei('d', x++, 10);
}
```

**getkey()**

The getkey() function waits for and returns the ASCII value of a pressed button.

*This function is only available on MT-40r1 and MT-40r2 chips - see MicroBoard documentation for details.*

**Example: key.prg**

```
// Demonstrates how to use the scankey() and getkey() functions on MicroBoard

#define PC      255
#define SYS    254
#define USER   253

main()
{
  int k;

  writes("key", 1000);
  loop {
    if(scankey()) {
```

```

    k = getkey();
    if('0' <= k && k <= '9') writec(0, k, 100);
    else
    if(k == PC) writes("pC", 100);
    else
    if(k == SYS) writes("sys", 100);
    else
    if(k == USER) writes("User", 100);
    } else writec(0, '.', 100);
}
}

```

**scankey()**

The scankey() function scans for and returns the ASCII value of a pressed button. It returns numeric value 0 (false) if no button is pressed. It does not wait for a button to be pressed as getkey() does.

*This function is only available on MT-40r1 and MT-40r2 chips - see MicroBoard documentation for details.*

**writes(s, T)**

The writes() function outputs the string 's' to the MicroBoard 7-Segments, MT Serial LCD or Serial COM port for 'T' milliseconds. 's' must either be a literal string enclosed in double quotes "" or a previously defined label for a string constant. To send the string to the serial port, precede the string with a '\$' character. To send a special command to MT Serial LCD Controller, precede the command string with a '~' character.

On the Microboard, if the length of the string is longer than the length of the display, then the rest of characters are shifted left one by one, for 'T' milliseconds. The shifting operation only applies to the 7-Segment display, not the serial port and MT Serial LCD.

```

#define MSG "This is great!" // a string constant
.....
writes(MSG, 1000); // o/p to 7-Segments or LCD
writes("Hello World", 1000); // o/p to 7-Segments or LCD
writes("~CLR", 0); // CLEAR LCD command
writes("$Hello World", 1000); // o/p to serial port

```

**puts(s)**

The puts() function outputs the *text string,s* to the serial port, without delay.

**writei(base, num, T)**

The writei() function outputs the *text string representation* of number 'num' of base 'base' to the MicroBoard 7-Segments, MT Serial LCD or Serial COM port for 'T' milliseconds.

Valid values for base are: 'd' = decimal, 'u' = unsigned, 'x' = hexadecimal, 'o' = octal and '232' = serial port. Only signed decimals ranging from -32768 to 32767 are supported for output to the serial port.

```

writei('x', 256, 1000); // displays 256 in hex number as 100
writei('o', 256, 1000); // displays 256 in octal number as 400
writei('u', 65535, 1000); // displays the maximum unsigned decimal integer
writei('d', 32767, 1000); // displays the maximum signed decimal integer
writei(232, 32767, 1000); // sends the maximum signed decimal integer to serial port

```

**putint(num)**

The putint() function outputs the *text string representation* of number 'num' of base 10 to the Serial COM. It is equivalent to writei(232, num, 0).

```

putint(32767); // sends the maximum signed decimal integer to serial port
delay(1000); // delay for 1s

```

**writec(digit, ch, T)**

The writec() function outputs the character 'ch' to the specified location 'digit' for 'T' milliseconds. If digit = 232 then writec() sends the character to the serial port.

```
writec(0, 'A', 1000); // displays character A at the 1st 7-Segment digit or LCD position
writec(232, 'A', 1000); // sends character A to serial port
```

**putchar(ch)**

The putchar() function sends the character to the serial port. It is equivalent to writec(232, ch, 0).

```
putchar('A'); // sends the character 'A' to serial port
delay(1000); // delay for 1s
```

**writeb(digit, ch, T)**

On the Microboard, the writeb() function outputs the unformatted byte 'ch' to the specified 'digit' for 'T' milliseconds. The bit pattern of the byte is clearly shown on the LED bar.

On MT chips, parameter 'digit' is ignored and the byte is output on I/O pins 0 - 7, with the most significant bit on pin 7. Use this function to output to devices that take a whole byte as data.

```
main()
{
    int i;

    i = 0xAA; // bits: 10101010
    loop {
        writeb(0, ~i, 1000); // unformatted byte output function
        i = ~i; // bits: 01010101
    }
}
```

**writem(type, addr, ch)**

The writem() function stores the 8-bit data 'ch' at the address location 'addr' of the specified memory storage device 'type' (EEPROM). Valid values for 'type' are: 24=24C02 and 93=93C56.

```
writem(24, 0, 'A'); // stores character A at the 1st location of EEPROM 24C02
writem(93, 10, 'A'); // stores character A at the 11th location of EEPROM 93C56
```

**Note:** EEPROM is a byte accessing device that there is no way to store a whole integer (say 12345) in a single byte storage unit. Users have to split the integer into 2 halves and store (later retrieve) them separately.

```
int x;

x = 12345;
writem(24, 0, x >> 8); // stores the upper 8 bits
writem(24, 1, x); // stores the lower 8 bits
```

**readm(type, addr)**

The readm() function retrieves the 8-bit data byte from address location 'addr' of the specified memory device 'type' (EEPROM). Valid values for 'type' are: 24=24C02, 93=93C56 and 232=serial port.

If 'type' = 232, then the 'addr' parameter is ignored and data is read from the serial port. If a character has been received then it is returned otherwise the function returns -1 (which indicating no character available). This, so called the 'non-blocking' feature is useful to test the availability of data on the serial port.

The following example shows how to retrieve an integer value previously saved by writem() function.

```
int x;
```

```
x = readm(24, 0) << 8; // retrieves the upper 8 bits
x += readm(24, 1);    // retrieves the lower 8 bits
```

The following example waits for a character on the serial port, and displays it's ASCII code.

```
int x;

do { x=readm(232, 0); } while(x == -1); // wait to receive a character
writei('d', x, 1000); // displays it's ASCII code
```

### getchar()

The `getchar()` function retrieves the 8-bit data byte from serial port. If a character has been received then it is returned otherwise the function returns `-1` (which indicating no character available). This, so called the 'non-blocking' feature is useful to test the availability of data on the serial port. It is equivalent to `readm(232, 0)`.

The following example waits for a character on the serial port, and displays it's ASCII code.

```
int x;

do { x=getchar(); } while(x == -1); // wait to receive a character
writei('d', x, 1000); // displays it's ASCII code
```

### setcomm(rate)

The `setcomm()` function opens and initialises the serial communication channel with baudrate set to 'rate'. Valid baudrates are: 0, 600, 1200, 2400, 4800, 9600 and 19200. A baudrate of 0 shuts down the serial port.

**Note:** The MT-08 and MT-14 chips only support 9600 baud and half-duplex mode. The `setcomm()` function is ignored and the communication channel is turned on as soon as the chips start running. Take care not to send data to them too fast to avoid overrunning their input buffers.

### delay(ms)

The `delay()` function halts program execution for 'ms' milliseconds.

### freemem()

The `freemem()` function returns the total number of free run-time memory units (in words) available.

### exit()

The `exit()` function terminates the program immediately.

### sys(type, value)

The `sys()` function provides the capability to set system variables during run-time and takes the following format: **sys(function type, parameter value);**

Function Category	1 <sup>st</sup> Parameter	2 <sup>nd</sup> Parameter	Description
COM port Selection	0	0..3	Select the current COM port
Timer Interrupt Operation	1	0	Switch timer Off
Timer Interrupt Operation	1	1..32767	Reset timer interval value
Day & Time (Year)	11	0..4095	Set year
Day & Time (Month)	12	1..12	Set month
Day & Time (Day)	13	1..28,29,30,31	Set day
Day & Time (Hours)	14	0..23	Set hours
Day & Time (Minutes)	15	0..59	Set minutes
Day & Time (Seconds)	16	0..59	Set seconds
Day & Time (Day of Week)	17	0..6	Set Sunday, Monday,..Saturday
Day & Time (Day of Year)	18	1..365,366	Set the N <sup>th</sup> day of year
Power Control	20	X	Put mcu into power down mode
Single PWM Mode	30	1	Enable Single PWM for 'servo()'
Single PWM Mode	30	0	Disable Single PWM for 'servo()'
DAC Operation	50	3001	Enable DAC
DAC Operation	50	3000	Disable DAC

DAC Operation	50	0..1023	Set DAC value
USB Operation	100	X	Enable USB COM ports

X = don't care, a negative pulse on the WU pin will wake up the chip previously put into power down mode, then it continues execution where it has left off. All functions are available on ARMBoards, but only functions with 1<sup>st</sup> parameters, 1, 20 and 30, are available on other types of MT chips.

### readsys(type)

The readsys() function provides the capability to read system variables during run-time and takes the following format: **readsys(function type);**

Function Category	1 <sup>st</sup> Parameter	Description
System Clock Ticks	10	Get system clock ticks in seconds (since startup)
Day & Time (Year)	11	Get year
Day & Time (Month)	12	Get month
Day & Time (Day)	13	Get day
Day & Time (Hours)	14	Get hours
Day & Time (Minutes)	15	Get minutes
Day & Time (Seconds)	16	Get seconds
Day & Time (Day of Week)	17	Get Sunday, Monday,..Saturday
Day & Time (Day of Year)	18	Get day of year

These functions are available on ARMBoards only.

### random()

The random() function generates and returns a pseudo-random number between 0 and 32767.

### sound(P, Hz, T)

The sound() function generates a square wave of approximate frequency of 'Hz' on I/O pin 'P' for 'T' milliseconds. Valid values for frequency range from 100 to 10,000. Since the main purpose of sound() is for the generation of musical notes (<2000Hz), the accuracy of the output frequency get deteriorates when set at a high value (>5KHz). This function can also be used to generate digital pulses for other circuits.

### readadc(P)

The readadc() function performs an A/D conversion on I/O pin 'P' and returns a value between 0 and 1023. The A/D conversion uses an internal reference of 5V (3.3V in ARM), so a value of '0' = 0 volts and a value of '1023' = 5V (3.3V in ARM) (see 'MT Chips Hardware Reference' - 'Analog-to-Digital Converter' section for details). This function is only available on MT-08, MT-14, MT-20, MT-28, MT-40-AD and MT ARM-based chips and only on specific pins that have ADC capability.

**Note:** The parameter, P refers to I/O Pin number, **NOT** the ADC channel number.

### readpwm(HL)

The readpwm() function returns the width (0 - 65535) in microseconds of a PWM signal present on the PWM pin. The measurement for the HIGH or LOW part of signal is specified by the HL parameter, 1 for High, 0 for Low.

### ircode()

The ircode() function is used to detect and decode IR codes sent in the APEX, NEC, PIONEER and HITACHI formats. It returns a value between 0 and 254 if successful or 255 if not.

### ps2key(), ps2x(), ps2y(), ps2h(), ps2v() and ps2(&ctrl, data)

The PS2 family of functions make life easier to detect and decode SONY or compatible PS2 game controllers, either wired or wireless versions. The MT chips firmware has been tested for more than 5 makes of controllers from various manufacturers, if your controller does not work properly, please obtain another one to try, or buy directly from MicroTech Systems Limited.

The ps2key() returns a character representing the detected key pressed. It is therefore impossible to return a combination of more than 2 keys being pressed simultaneously. However, user can simply use the ps2() function to retrieve the raw packets and decode them manually. The other four function returns the values of the 4 axis of that 2 controller sticks.

In function ps2(), raw packets returned which indicate the type of PS2 Controller, buttons settings and the current positions of the 2 sticks.

Do study the given sample programs, “ps2key.prg”, “ps2.prg”, “ps2-b.prg” and “ps2sticks.prg” for demonstrations. Do make sure that the PS2 Controller is correctly connected, otherwise an error message will be displayed.

Note: 10K pull-up resistors are required for MT-20, MT-28 and MT-40 types of chips, ARMBBoard not required.

### servo(P, deg)

The servo() function generates a chain positive pulses (see note) of width from 0.3ms to 2.8ms to control RC servo motors. The pulses are output on pin 'P' and 'deg' is the relative degree (0 - 180) of rotation. The width of the pulse is designed to cater for a wide ranges of different servos. Care must be taken **NOT** to overdrive servos to their extreme limits at both ends. Start with 90° first, then work out the upper and lower limits the servo can take. Otherwise, the gears inside the servo might easily get damaged due to overdriving. Since RC servos draw a lot current when they move, therefore, it is recommended a separate power source (Ni-Cad, Ni-MH or Li-Ion batteries) be supply to power the servos, with the negative wire connects to the **Ground** of the MT chip as the common signal **GND** reference.

The degree of throw may vary between different types of servos. This system is designed based on standard sized RC servos, like Futaba 3001 and 3003, which are capable of throws of more than 180 degrees. The smaller ones, like Futaba 3106, have smaller degrees of throws.

It is possible to move several servos simultaneously. In this case simply add 100 to the pin numbers, then issue the "move all" servo command, i.e. “servo(200, 0);”, with 200 as its pin number. See example programs ‘servo-3.prg’ and ‘servo-5.prg’.

The MT-40 series chips are able to support up to 32 servos on their I/O pins, but, in order to move servos quickly and smoothly, a maximum of 10 servos is recommended.

**Note:** The servo() function will stop emitting pulses after it's execution. In order to send continuous controlling pulses to servos to maintain their holding power, a dedicated servo controller (like MT-SC21 or MT-SC13) must be deployed. The servo() can be set to emit continuous PWM pulses by using the sys(30, 1) function. See example, “adc-servo.prg” for demonstration.

### Example: servo-3.prg

```
// Control the movements of 2 RC servos

main()
{
  loop {
    // move servos individually!
    servo(3, 20);
    delay(1000);
    servo(3, 160);
    servo(4, 20);
    delay(1000);
    servo(4, 160);
    delay(1000);

    // move servos simultaneously!
    servo(103, 20); // set angle
    servo(104, 20); // set angle
    servo(200, 0); // start moving all

    delay(1000);

    // move servos simultaneously!
    servo(103, 160); // set angle
    servo(104, 160); // set angle
    servo(200, 0); // start moving all

    delay(1000);
  }
}
```

## Using I/O Pin External Interrupts

An external interrupt function is a special kind of function which is called when it's associated I/O pin is activated ie. pulled LOW. When the interrupt function finishes, the program resumes execution from the point where the interrupt was activated.

**Syntax:** *function\_name()* *interrupt* <constant expression> { ..... }

The following defines say\_hello() to be an interrupt function associated with I/O pin 1. It is activated when I/O pin 1 goes LOW (0V).

```
say_hello() interrupt 1
{
  writes("Hello", 1000);
}
```

I/O interrupts in TinyC are not hardware interrupts. The I/O pins are checked (polled) between execution of each instruction in the program. Therefore, statements that take a while to finish eg. writes("hello", 5000) will slow down the response to the interrupt. In this case the interrupt won't be acknowledged for 5 seconds.

In order to use external interrupts, the following rules must be observed:

- Only up to 5 interrupt functions can be defined in a program.
- Apart from the main() function, any function can be an interrupt function.
- Each interrupt function must be associated with an I/O pin.
- Interrupt functions cannot receive or return values.
- An interrupt function cannot call itself or another interrupt function but can call any other normal function.
- Interrupt functions can be called by any other normal functions anywhere within the program just like normal function calls. However this is a bad programming practice and is not encouraged. Use it mainly for testing purposes.
- Only one interrupt function is activated at any given time ie. interrupts within interrupts are not allowed.

To disable the interrupt capability of an interrupt function, just delete the "interrupt" keyword and the associated pin number. The function then reverts back to a normal function.

The interrupt function will be re-activated if the associated pin is still LOW when the interrupt function exits. If this is a problem then it is up to the user program to test and wait for the associated I/O pin to go HIGH before exiting the interrupt function.

## Using the Timer Interrupt

TinyC provides a timer interrupt function. This makes tasks such as refreshing the data on a 7-segment display in the background much easier. Without a timer interrupt this would be almost impossible. The timing interval is specified in milliseconds from 1 to 32767 allowing a timed interrupt every 32.767 seconds if set to the maximum.

**Syntax:** *function\_name()* *timer* <constant expression> { ..... }

The following example defines 'message()' to be a timer function and sets the timer interrupt at 30000 milliseconds, and is called every 30000ms (30 seconds) and displays the message.

```
message() timer 30000
{
  writes("Hello, see you again later!", 1000);
}
```

The system management function **sys()** can be used to change the timing value, as follows:

```
sys(1, 10000);    // reset to 10000ms (10 second) intervals
sys(1, 0);        // setting = 0 means stop (switch-off) the timer! No more timer interrupts.
                  // The first parameter refers to the timer.
                  // The second parameter sets the timing interval.
```

**NOTE:** Whenever the timer interrupt function is present and **switch-on** in a program, the `sound()` function for the following chips is disabled. This is because both functions use the same hardware interrupt in these chips. However, as soon as the timer function is switched off, the `sound()` function will be enabled again.

MT-40r1, MT-40r2, MT-20, MT-28

## Developing TinyC programs for MT chips

The software package used for developing TinyC programs for MT Chips is called the MicroTech Editor. It provides an Integrated Development Environment (IDE) with built in text editor and downloader. The IDE invokes an external compiler program, 'cmp.exe', to compile the TinyC programs. The downloader transfers the compiled program to the MT chip.

An external downloader, 'Downloader.exe'. is also included in the package. Users can integrate these two commands into their favourite programming editor for compiling and downloading programs. Note that 'Downloader.exe' takes the 'COM port' and 'Delay timing' parameter settings from 'Microtech.ini' file.

The MicroTech IDE also has a built-in RS232 terminal program for testing serial devices connected to a PC COM port. It can be used to send commands and data to an MT serial LCD unit (configured for RS232 mode) and to test communication links with MT chips.

The MicroTech IDE software is freely available and the latest version can be downloaded from the MicroTech website at [www.mcu.hk](http://www.mcu.hk).

Developing TinyC programs for MT chips requires the following items:

- PC running Microsoft Windows 98, 2000, XP, etc
- MicroTech Editor software package
- MT microcontroller chip
- MT demo board or a breadboard/PCB with a RS232 circuitry
- 5V DC power supply
- MT serial download cable
- MT serial LCD unit (optional but highly recommended)

The MT serial LCD unit is strongly recommended for displaying messages when downloading and running programs. The supplied serial download cable has a 9-pin 'D' connector for direct connection to a PC COM port. If your computer does not have this type of connector then a USB to serial adapter/converter is needed.

## TinyC Software Installation

To install TinyC you require a computer running Windows 98 or later with approximately 5MB free space.

- Download and unzip MT software pack to the hard disk.
- Create a shortcut to the editor software, 'Microtech.exe', and place it on the desktop for easy access.
- Run the Editor.
- Click 'File → Options' to change the optional settings.
- The most important setting here is the COM port number. The software will automatically display all the available COM ports on the computer. Select the COM port that will be used for downloading.
- Change the 'Delay Timing' to adjust the downloading speed. The smaller the number the faster the download speed. However setting too fast a speed will corrupt the download process.
- Change the 'Working Directory' to the directory where the sample programs are located. Click the 'Save Settings' button to save the new settings.

## Developing TinyC programs

- The MicroTech Editor invokes the external program, 'cmp.exe' to compile programs.
- TinyC programs are simply text files - any text editor can be used.
- Right clicking anywhere on the text will invoke the popup command menu.
- Compile the program.
- Programs are saved automatically before being compiled.
- Download the program to the MT chip using the built-in downloader.
- There is an external command line downloader, 'Downloader.exe', for transferring programs into MT chips, and it takes parameter settings from the 'Microtech.ini' file. 'cmp.exe' and 'Downloader.exe' are

provided so that users can integrate them into their preferred programming editors for program development. Output messages from these commands are stored in files CMP.TMP and DL.TMP.

## Running programs using the MT chips Simulator

The Simulator enables users to develop and test programs without the need for any hardware. It does not provide any debugging tools such as breakpoints or single stepping but it does support the various MT chip I/O functions.

1. Compile the TinyC program.
2. Click on the Simulator button on the Editor menu bar. You will be prompted for the name of the program to simulate (defaults to the current file). Select the file and click 'Open'.
3. Click the 'Run' button to run the program.
4. Note that, the maximum code size that the Simulator can handle is about 4K words.

The timings and delays in the simulator are different to those in real MT chips. The simulator is used to show that a TinyC program is running and working.

All the TinyC functions are simulated, including the 5 I/O pin interrupts, the timer interrupt (partly) and reading and writing data to serial EEPROMs. The data for the EEPROMs are stored in the 24C02.DAT and 93C56.DAT files.

Press the buttons within the Demo box to see a demonstration of the various functions. They are disabled when a TinyC program is running, and re-enabled when the program is halted by the STOP button.

Click the I/O Pin boxes to set the pin states. If the box is checked then the output is low (0V).

Use the Sliding Bar to set values for the readadc() function.

Use the keypad (bottom-left) to input values for ircode(), getkey(), scankey() and serial readm() functions. The SYS and USER buttons are tied to I/O pins 19 and 18, respectively. They toggle the pin state each time they are clicked. **Note:** On a real MicroBoard, these 2 keys are activated only when they are kept pressed down.

Output to serial port by all the writeX() functions goes to the COM Output box.

The values shown in the Sound and Servo boxes are the I/O pin numbers, frequencies and degrees for sound() and servo() functions.

Function writec() does not clear the LCD before or after displaying data.

The MT Serial LCD unit controlling commands are not simulated yet!

Click on the Exit button to save the EEPROM data to disk files and exit the simulator.

## Downloading TinyC programs for MT chips with single storage area

- Connect the download cable to a COM port on the PC.
- Pull the PC pin (bottom right pin) LOW (connect to ground).
- Switch on or press the Reset button on the Demo Board or User breadboard.
- The MT-40-AD chip needs a negative pulse for reset. There is no reset pin on MT-08 and MT-14 chips. The only way to reset them is to power off then on again.
- The "PC" message will be displayed on the serial LCD if connected.
- Click the Download button on the Editor menu bar.
- After a successful download, the message, "OK" appears. Soon after, another message, "Run" appears.
- A number is displayed on the serial LCD indicating the amount of free run-time memory units available.
- Program starts running.

## Downloading TinyC programs for MT chips with multiple storage areas

- Connect the download cable to a COM port on the PC.
- Pull the PC pin (bottom right pin) LOW.
- By default the program is downloaded to storage area 0. To use another storage area pull the desired program storage area pin LOW at the same time as the PC pin.
- Switch on or press the reset button on the Demo Board.
- The "PC-" message will be displayed on the LCD.
- Click the Download button on the Editor menu bar.
- The "PC-x" message will be displayed on the LCD, where x is the selected storage area.
- After a successful download, the message, "OK" appears. Soon after, another message, "Run-x" appears.
- A number appears indicating the amount of free run-time memory units.
- Program starts running.

### Tips:

- Keep the program storage area pin LOW to select it for running.
- The PC pin needs to keep LOW only for downloading purpose.
- The default program storage area and 'program to run' is 0.
- Instead of setting the I/O pins manually, compiler directives can be used to specify the storage area to store the program and the default program to run after reset. However, the PC pin still be pulled LOW to initiate the download process.

## Downloading TinyC programs for MicroBoard

- Connect the download cable to a COM port on the PC.
- Switch on or press the reset button on the MicroBoard.
- Press the PC button until message, "PC 0-4" appears then press button 0 - 4 to select the program storage area. Wait until the selected digit disappears on the display before starting to download.
- Click the Download button on the Editor menu bar.
- The red and green LEDs on the MicroBoard will flash as the program is downloaded.
- After a successful download, the message, "Done" appears.
- Soon after, the message "Run 0-4" appears. Press button 0 - 4 to select the program to run.
- A number appears indicating the amount of free run-time memory units.
- Program starts running.

**Note:** During download processes described above the error message, "Size Error", will be displayed if the size of the program is too large for the program storage area. Another error message, "Version Error", will be displayed if the firmware of the MT chip is not compatible with the version of compiler.

## Running TinyC programs for MicroBoard

- Switch on or press the reset button on the MicroBoard.
- Wait until message "Run 0-4" appears then press button 0 - 4 to select the program to run.
- A number appears indicating the amount of free run-time memory units.
- Program starts running.