

PIC Simulator IDE

BASIC Compiler Reference Manual

Table Of Contents:

General info

[Show Warnings](#), [Do Not Compile Unused Code](#), [Initialize Variables On Declaration](#),

About variables

[Dim](#), [As](#), [Bit](#), [Byte](#), [Word](#), [Long](#), [Reserve](#), [.HB](#), [.LB](#), [.HW](#), [.LW](#), [Symbol](#), [True](#), [False](#), [Const](#), [Pointer](#), [ASM](#), [WREG](#),

Mathematical and logical operations

[Mod](#), [Sqr](#), [Not](#), [And](#), [Or](#), [Xor](#), [Nand](#), [Nor](#), [Nxor](#), [High](#), [Low](#), [Toggle](#),

Standard Basic language elements

[Goto](#), [For](#), [To](#), [Step](#), [Next](#), [While](#), [Wend](#), [If](#), [Then](#), [Else](#), [Endif](#), [Select Case](#), [Case](#), [EndSelect](#), [Halt](#),

Subroutines

[End](#), [Gosub](#), [Return](#),

Microcontroller related language elements

[Config](#), [Input](#), [Output](#), [AllDigital](#), [Define](#), [CONF_WORD](#), [CONF_WORD_2](#), [CLOCK_FREQUENCY](#), [EEPROM](#),

Special Basic language elements

[WaitMs](#), [WaitUs](#), [SIMULATION_WAITMS_VALUE](#), [Break](#), [ShiftLeft](#), [ShiftRight](#), [LookUp](#), [Count](#), [COUNT_MODE](#), [FreqOut](#),

Structured language support (procedures and functions)

[Proc](#), [End Proc](#), [Call](#), [Exit](#), [Function](#), [End Function](#), [Include](#),

Using internal EEPROM memory

[Read](#), [Write](#),

Using internal A/D converter module

[Adcin](#), [ADC_CLOCK](#), [ADC_SAMPLEUS](#),

Using interrupts

[Enable](#), [Disable](#), [On Interrupt](#), [Resume](#), [Save System](#),

Serial communication using internal hardware UART

[Hseropen](#), [Hserout](#), [Hserin](#), [Hserget](#), [CrLf](#), [Lf](#), [ALLOW_MULTIPLE_HSEROPEN](#), [ALLOW_ALL_BAUDRATES](#),

Software UART implementation

[SEROUT_DELAYUS](#), [Serout](#), [Serin](#), [SeroutInv](#), [SerinInv](#),

Interfacing character LCDs

[LCD_BITS](#), [LCD_DREG](#), [LCD_DBIT](#), [LCD_RSREG](#), [LCD_RSBIT](#), [LCD_EREG](#), [LCD_EBIT](#), [LCD_RWREG](#), [LCD_RWBIT](#), [LCD_COMMANDUS](#), [LCD_DATAUS](#), [LCD_INITMS](#), [LCD_READ_BUSY_FLAG](#), [Lcdinit](#), [Lcdout](#), [Lcdcmdout](#), [LcdClear](#), [LcdHome \(LcdLine1Home\)](#), [LcdLine2Home](#), [LcdDisplayOn](#), [LcdDisplayOff](#), [LcdCurOff](#), [LcdCurBlink](#), [LcdCurUnderline](#), [LcdCurBlinkUnderline](#), [LcdLeft](#), [LcdRight](#), [LcdShiftLeft](#), [LcdShiftRight](#), [LcdLine1Clear](#), [LcdLine2Clear](#), [LcdLine1Pos](#), [LcdLine2Pos](#), [Lcddefchar](#), [LcdLine3Home](#), [LcdLine4Home](#), [LcdLine3Clear](#), [LcdLine4Clear](#), [LcdLine3Pos](#), [LcdLine4Pos](#), [LCD_LINES](#), [LCD_CHARS](#),

I2C communication with external I2C devices

[I2CWrite](#), [I2CRead](#), [I2CWrite1](#), [I2CRead1](#), [I2CREAD_DELAYUS](#), [I2CCLOCK_STRETCH](#),
[I2CPrepare](#), [I2CStart](#), [I2CStop](#), [I2CSend](#), [I2CRecA \(I2CReceiveAck\)](#), [I2CRecN \(I2CReceiveNack\)](#),

Support for Serial Peripheral Interface (SPI) communication

[SPI_SCK_REG](#), [SPI_SCK_BIT](#), [SPI_SDI_REG](#), [SPI_SDI_BIT](#), [SPI_SDO_REG](#), [SPI_SDO_BIT](#),
[SPI_CS_REG](#), [SPI_CS_BIT](#), [SPICLOCK_INVERT](#), [SPICS_INVERT](#), [SPICLOCK_STRETCH](#),
[SPIPrepare](#), [SPICSON](#), [SPICSOFF](#), [SPISend](#), [SPIReceive](#), [SPISendBits](#),

Interfacing graphical LCDs with 128x64 dot matrix

[GLCD_DREG](#), [GLCD_RSREG](#), [GLCD_RSBIT](#), [GLCD_EREG](#), [GLCD_EBIT](#), [GLCD_RWREG](#),
[GLCD_RWBIT](#), [GLCD_CS1REG](#), [GLCD_CS1BIT](#), [GLCD_CS2REG](#), [GLCD_CS2BIT](#), [GLcdinit](#),
[GLcdclear](#), [GLcdpset](#), [GLcdpreset](#), [GLcdclean](#), [GLcdposition](#), [GLcdwrite](#), [GLcdout](#), [GLcdin](#),
[GLcdcmdout](#),

Using internal PWM modules

[PWMon](#), [PWMoff](#), [PWMduty](#),

Interfacing Radio Control (R/C) servos

[Servoin](#), [ServoOut](#),

Interfacing Stepper Motors

[STEP_A_REG](#), [STEP_A_BIT](#), [STEP_B_REG](#), [STEP_B_BIT](#), [STEP_C_REG](#), [STEP_C_BIT](#),
[STEP_D_REG](#), [STEP_D_BIT](#), [STEP_MODE](#), [StepHold](#), [StepCW](#), [StepCCW](#),

Interfacing 1-WIRE devices

[1WIRE_REG](#), [1WIRE_BIT](#), [1wireInit](#), [1wireSendBit](#), [1wireGetBit](#), [1wireSendByte](#), [1wireGetByte](#),
[DS18S20Start](#), [DS18S20ReadT](#),

Advanced features

[StartFromZero](#),

• General info

Basic compiler editor is composed of editor panel (for user program editing) and source explorer (for easy navigation through all elements of user program - variables, symbols, constants, subroutines, procedures and functions). Editor formats and colorizes entered lines of user program, that simplifies the debugging process.

The primary output of the compiler is an assembler source file. However, with an appropriate command from the menu it can be assembled and even loaded in the simulator with a single click. Menu commands and options are rich, as well as the commands from the right-click popup menus for the editor and source explorer. Basic compiler's assembler output contains many useful comment lines, that makes it very helpful for educational purposes, also.

Show Warnings

If Show Warnings option is enabled, in the Warnings window Basic compiler will show information about unused declarations, subroutines, procedures and functions in the user basic program.

Do Not Compile Unused Code

If this option is enabled, Basic compiler will not compile unused declarations, subroutines, procedures and functions, in order to save memory resources.

Initialize Variables On Declaration

If this option is enabled, Basic compiler will reset to zero all memory locations allocated for variables, at the position of their declaration in the basic program. This option is useful for beginners, because RAM memory is filled with random values at device power-up, and it is easy to make a mistake to assume that all variables are reset to zero at power-up. Experienced users can save some program memory, by disabling this option and taking control of variable initial values by user program where necessary.

- [About variables](#)

Four data types are supported:

[Bit](#) - 1-bit, 0 or 1

[Byte](#) - 1-byte integers in the range 0 to 255

[Word](#) - 2-byte integers in the range 0 to 65,535

[Long](#) - 4-byte integers in the range 0 to 4,294,967,295

Variables can be global (declared in the main program, before the End statement) or local (declared in subroutines, procedures and functions). Variable name used for a variable with global scope can be used again for local variable names. The compiler will reserve separate memory locations for them. The total number of variables is limited by the available microcontroller RAM memory.

Variables are declared using DIM statement:

```
Dim i As Bit
Dim j As Byte
Dim k As Word
Dim x As Long
```

If necessary, variable address can be specified during declaration:

```
Dim x As Byte @ 0x050
```

It is possible to use one-dimensional arrays for Byte and Word variables. For example:

```
Dim x(10) As Byte
```

declares an array of 10 Byte variables with array index in the range [0-9].

Long data type (32-bit math support) is an optional add-on module for the integrated basic compiler.

RESERVE statement allows advanced usage by reserving some of the RAM locations to be used by in-code assembler routines or by MPLAB In-Circuit Debugger. For example:

```
Reserve 0x70
```

High and low byte of a word variable can be addressed by .HB and .LB extensions. Individual bits can be addressed by .0, .1, ..., .14 and .15 extensions. It is possible to make conversions between Byte and Word data types using .LB and .HB extensions or directly:

```
Dim x As Byte
Dim y As Word
x = y.HB
x = y.LB 'This statement is equivalent to x = y
y.HB = x
y.LB = x
y = x 'This statement will also clear the high byte of y variable
```

High word (composed by bytes 3 and 2) and low word (composed by bytes 1 and 0) of a long variable can be addressed by .HW and .LW extensions. Byte 0 can be addressed by .LB and byte 1 by .HB extensions. For example:

```
Dim i As Byte
Dim j As Word
Dim x As Long
i = x.LB
j = x.HW
```

All special function registers (SFRs) are available as Byte variables in basic programs. Individual bits of a Byte variable can be addressed by .0, .1, .2, .3, .4, .5, .6 and .7 extensions or using official names of the bits:

```
Dim x As Bit
Dim y As Byte
x = y.7
y.6 = 1
TRISA.1 = 0
TRISB = 0
```

```

PORTA.1 = 1
PORTB = 255
STATUS.RP0 = 1
INTCON.INTF = 0

```

Standard short forms for accessing port registers and individual chip pins are also available (RA, RB, RC, RD, RE can be used as Byte variables; RA0, RA1, RA2, ..., RE6, RE7 are available as Bit variables):

```

RA = 0xff
RB0 = 1

```

It is possible to use symbolic names (symbols) in programs, to easily address system variables. Symbols can be global or local. SYMBOL directive is used to declare symbolic names:

```

Symbol led1 = PORTB.0
led1 = 1
Symbol ad_action = ADCON0.GO_DONE

```

Constants can be used in decimal number system with no special marks, in hexadecimal number system with leading 0x notation (or with H at the end) and in binary system with leading % mark (or with B at the end). Keywords True and False are also available for Bit type constants. For example:

```

Dim x As Bit
Dim y As Byte
x = True
y = 0x55
y = %01010101

```

Constants can be assigned to symbolic names using CONST directive. Constants can be global or local. One example:

```

Dim x As Word
Const pi = 314
x = pi

```

Any variable that is declared as a Byte or Word variable using Dim statement can be used as a pointer to user RAM memory when it is used as an argument of POINTER function. The value contained in the variable that is used as a pointer should be in the range 0-511. Here is one example:

```

Dim x As Word
Dim y As Byte
x = 0x3f
y = Pointer(x)
y = y + 0x55
x = x - 1
Pointer(x) = y
y = 0xaa
x = x - 1
Pointer(x) = y

```

It is possible to use comments in basic source programs. The comments must begin with single quote symbol (') and may be placed anywhere in the program.

Lines of assembler source code may be placed anywhere in basic source program and must begin with ASM: prefix. For example:

```

ASM:      NOP
ASM:LABEL1: MOVLW 0xFF

```

Symbolic names of declared variables can be used in assembler routines because proper variable address will be assigned to those names by EQU directive:

```

Dim varname As Byte
varname = 0
ASM:      MOVLW 0xFF
ASM:      MOVWF VARNAME

```

When working with inline assembler code, it could be useful to use working register as a source or destination in assign statements. For that purpose WREG keyword should be used and the compiler will take care of the bank control:

```
Dim varname As Byte
ASM:      MOVLW 0xFF
varname = WREG
```

• Mathematical and logical operations

Five arithmetic operations (+, -, *, /, MOD) are available for Byte, Word and Long data types. The compiler is able to compile all possible complex arithmetic expressions. For example:

```
Dim i As Word
Dim j As Word
Dim x As Word
i = 123
j = i * 234
x = 2
x = (j * x - 12345) / (i + x)
```

Square root of a number (0-65535 range) can be calculated using SQR function:

```
Dim x As Word
x = 3600
x = Sqr(x)
```

For Bit data type variables seven logical operations are available. It is possible to make only one logical operation in one single statement. Logical operations are also available for Byte and Word variables. For example:

Example 1:

```
Dim i As Bit
Dim j As Bit
Dim x As Bit
x = Not i
x = i And j
x = i Or j
x = i Xor j
x = i Nand j
x = i Nor j
x = i Nxor j
```

Example 2:

```
Dim x As Word
Dim y As Word
x = x Or y
PORTB = PORTC And %11110000
```

There are three statements that are used for bit manipulation - HIGH, LOW and TOGGLE. If the argument of these statements is a bit in one of the PORT registers, then the same bit in the corresponding TRIS register is automatically cleared, setting the affected pin as an output pin. Some examples:

```
High PORTB.0
Low ADCON0.ADON
Toggle OPTION_REG.INTEDG
```

• Standard Basic language elements

Unconditional jumps are performed by GOTO statement. It uses line label name as argument. Line labels can be global or local. Line labels must be followed by colon mark ':'. Here is one example:

```
Dim x As Word
x = 0
loop: x = x + 1
```

Goto loop

Four standard BASIC structures are supported: FOR-TO-STEP-NEXT, WHILE-WEND, IF-THEN-ELSE-ENDIF and SELECT CASE-CASE-ENDSELECT. Here are several examples:

Example 1:

```
Dim x As Byte
TRISB = 0
x = 255
while x > 0
    PORTB = x
    x = x - 1
    waitMs 100
wend
PORTB = x
```

Example 2:

```
TRISB = 0
loop:
    If PORTA.0 Then
        PORTB.0 = 1
    Else
        PORTB.0 = 0
    Endif
    Goto loop
```

Example 3:

```
Dim x As Word
TRISB = 0
For x = 0 To 10000 Step 10
    PORTB = x.LB
Next x
```

Example 4:

```
Dim i As Byte
Dim j As Byte
Dim x As Byte
j = 255
x = 2
TRISB = 0
For i = j To 0 Step -x
    PORTB = i
Next i
```

Example 5:

```
Dim x As Byte
loop:
    Select Case x
        Case 255
            x = 1
        Case <= 127
            x = x + 1
        Case Else
            x = 255
        EndSelect
    Goto loop
```

After IF-THEN statement in the same line can be placed almost every other possible statement and then ENDIF is not used. There are no limits for the number of nested statements of any kind. In the test expressions of IF-THEN and WHILE statements it is possible to use multiple ORed and multiple ANDed conditions. Multiple comma separated conditions can be used with CASE statements, also.

If there is a need to insert an infinite loop in basic program, that can be done with HALT statement.

- **Subroutines**

Structured programs can be written using subroutine calls with GOSUB statement that uses line label name as argument. Return from a subroutine is performed by RETURN statement. User need to take care that the program structure is consistent. When using subroutines, main routine need to be ended with END statement. END statement is compiled as an infinite loop. Here is an example:

```
Symbol ad_action = ADCON0.GO_DONE
Symbol display = PORTB
TRISB = %00000000
TRISA = %111111
ADCON0 = 0xc0
ADCON1 = 0
High ADCON0.ADON
main:
  Gosub getadresult
  display = ADRESH
  Goto main
End

getadresult:
  High ad_action
  while ad_action
  wend
  Return
```

- **Microcontroller related language elements**

Microcontroller ports and pins can be configured as inputs or outputs by assigning proper values to TRISx registers or their bits. That task can also be accomplished by a CONFIG statement. Its syntax is apparent from the following examples:

```
Config PORTB = Output
Config RA0 = Output
Config PORTC.3 = Input
Config RD = Input
```

All PIC microcontrollers that feature analog capabilities (A/D converters and/or analog comparators) are setup at power-up to use the involved pins for these analog purposes. In order to use those pins as digital input/outputs, they should be setup for digital use by changing the values in some of the special functions registers as specified by the datasheets. To setup all pins for digital purposes, ALLDIGITAL statement can be used at the beginning of the basic program.

There are two configuration parameters CONF_WORD and CONF_WORD_2 (not available for all devices) that can be set using DEFINE directive to override the default values. The clock frequency of the target device can be specified by setting the CLOCK_FREQUENCY parameter (the value is expressed in MHz). These parameters should be setup at the beginning of the basic program. For example:

```
Define CONF_WORD = 0x3f72
Define CLOCK_FREQUENCY = 20
```

EEPROM memory content can be defined in basic programs using EEPROM statement. Its first argument is the address of the first byte in the data list. Multiple EEPROM statements can be used to fill in different areas of EEPROM memory, if needed. For example:

```
EEPROM 0, 0x55
EEPROM 253, 0x01, 0x02, 0x03
```

- **Special Basic language elements**

WAITMS and WAITUS statements can be used to force program to wait for the specified number of milliseconds or microseconds. It is also possible to use variable argument of Byte or Word data

type. These routines use Clock Frequency parameter that can be changed from the Options menu. WAITUS routine has minimal delay and step that also depend on the Clock Frequency parameter.

```
Dim x As Word
x = 100
waitMs x
waitUs 50
```

Important Note: When writing programs for real PIC devices you will most likely use delay intervals that are comparable to 1 second or 1000 milliseconds. Many examples in this help file also use such 'real-time' intervals. But, if you want to simulate those programs you have to be very patient to see something to happen, even on very powerful PCs available today. For simulation of 'WaitMs 1000' statement on 4MHz you have to wait the simulator to simulate 1000000 instructions and it will take considerable amount of time even if 'extremely fast' simulation rate is selected. So, just for the purpose of simulation you should recompile your programs with adjusted delay intervals, that should not exceed 1-10ms. But, be sure to recompile your program with original delays before you download it to a real device. There is an easy way to change arguments of all WAITMS statements in a large basic program with a value in the range 1-10 for simulation purposes. With one line of code setting parameter SIMULATION_WAITMS_VALUE with DEFINE directive, the arguments of all WAITMS statements in the program will be ignored and the specified value will be used instead during compiling. Setting the value 0 (default) for this parameter (or omitting the whole line) will cancel its effect and the compiled code will be ready again for the real hardware.

It is possible to insert breakpoints for the simulator directly in basic programs using BREAK statement. It is compiled as reserved opcode 0x0001 and the simulator will interpret this opcode as a breakpoint and switch the simulation rate to Step By Step.

SHIFLEFT and SHIFTRIGHT functions can be used to shift bit-level representation of a variable left and right. The first argument is input variable and the second argument is number of shifts to be performed. Here are two examples:

Example 1:

```
TRISB = 0x00
PORTB = %00000011

goleft:
    waitMs 250
    PORTB = ShiftLeft(PORTB, 1)
    If PORTB = %11000000 Then Goto goright
    Goto goleft

goright:
    waitMs 250
    PORTB = ShiftRight(PORTB, 1)
    If PORTB = %00000011 Then Goto goleft
    Goto goright
```

Example 2:

```
TRISB = 0x00
PORTB = %00000001

goleft:
    waitMs 250
    PORTB = ShiftLeft(PORTB, 1)
    If PORTB.7 Then Goto goright
    Goto goleft

goright:
    waitMs 250
    PORTB = ShiftRight(PORTB, 1)
    If PORTB.0 Then Goto goleft
    Goto goright
```


LOOKUP function can be used to select one from the list of Byte constants, based on the value in the index Byte variable, that is supplied as the last separated argument of the function. The first constant in the list has index value 0. The selected constant will be loaded into the result Byte data type variable. If the value in the index variable goes beyond the number of constants in the list, the result variable will not be affected by the function. Here is one small example for a 7-segment LED display:

```
Dim digit As Byte
Dim mask As Byte
TRISB = %00000000
loop:
  For digit = 0 To 9
    mask = Lookup(0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f,
0x6f), digit
    PORTB = mask
    WaitMs 1000
  Next digit
  Goto loop
```

If all constants in the list (or part of them) are ASCII values, then shorter form of the list can be created by using string arguments. For example:

```
mask = Lookup("ABCDEFGHIJK"), index
```

If it is necessary to count the number of pulses that come to one of the microcontroller's pins during a certain period of time, there is COUNT statement available for that purpose. It has three arguments. The first one is the pin that is connected to the source of pulses. It should previously be setup as digital input pin. The second argument defines the duration of the observation expressed in milliseconds and it must be a numeric constant in the range 1-10000. The last argument of this statement is a Byte or Word variable where the counted number of pulses will be stored after its execution. COUNT statement uses internal Timer0 peripheral module. There is COUNT_MODE parameter available that can be setup with DEFINE directive. If it is set to value 1 (default value) COUNT statement will count the number of rising pulse edges. If COUNT_MODE = 2, the number of falling edges will be counted.

```
Define COUNT_MODE = 1
Dim num_of_pulses As Word
Count PORTB.0, 1000, num_of_pulses
```

FREQOUT statement can be used to generate a train of pulses (sound tone) on the specified pin with constant frequency and specified duration. It has three arguments. The first argument is the pin that the tone will be generated on. It should previously be setup as digital output pin. The second argument specify the tone frequency and it must be a constant in the range 1-10000Hz. The third argument defines the tone duration and it also must be a numeric constant in the range 1-10000ms. Choosing higher tone frequencies with low microcontroller clock frequency used may result in somewhat inaccurate frequency of the generated tones. FREQOUT statement can be alternatively used in 'variable mode' with Word data type variables instead of constants for the last two arguments. In this mode of usage the second argument is supposed to hold the half-period of the tone (in microseconds) and the third argument must hold the total number of pulses that will be generated. The following code will generate one second long tone on RB0 pin with 600Hz frequency:

```
TRISB.0 = 0
FreqOut PORTB.0, 600, 1000
```

• Structured language support (procedures and functions)

Procedures can be declared with PROC statement. They can contain up to 5 arguments (comma separated list) and all available data types can be used for argument variables. Argument variables are declared locally, so they do not need to have unique names in relation to the rest of user basic program, that makes very easy to re-use once written procedures in other basic programs. The procedures can be exited with EXIT statement. They must be ended with END PROC statement and must be placed after the END statement in program. Calls to procedures are implemented with CALL statement. The list of passed arguments can contain both variables and numeric constants. For example:

```

Dim x As Byte
TRISB = 0
For x = 0 To 255
    Call portb_display(x)
    WaitMs 100
Next x
End

Proc portb_display(arg1 As Byte)
PORTB = arg1
End Proc

```

All facts stated for procedures are valid for functions, also. Functions can be declared with FUNCTION statement. They can contain up to 5 arguments and argument variables are declared locally. Functions can be exited with EXIT statement and must be ended with END FUNCTION. The name of the function is declared as a global variable, so if the function is called with CALL statement, after its execution the function variable will contain the result. Standard way of function calls in assignment statements can be used, also. One simple example:

```

Dim x As Byte
Dim y As Word
For x = 0 To 255
    y = square(x)
Next x
End

Function square(arg1 As Word) As Word
square = arg1 * arg1
End Function

```

Basic source code from an external file can be included to the current program by using INCLUDE directive. Its only argument is a string containing the path to the external .BAS file. This can be the full path or only the file name, if the external file is located in the same folder as the current basic program file. During the compilation process the external basic source will be appended to the current program. Multiple files can be included with separate INCLUDE directives. To maintain the overall basic code structure, it is strongly suggested that the external file contains global declarations, subroutines, procedures and functions, only. Here is one very simple example for the demonstration:

```

main.bas:
Dim i As Word
Dim j As Word

Include "inc1.bas"
Include "inc2.bas"

For i = 1 To 10
    j = func1(i, 100)
    Call proc1(j)
Next i
End

inc1.bas:
Dim total As Word

Proc proc1(i As Word)
total = total + i
End Proc

inc2.bas:
Function func1(i As Word, j As Word) As Word
func1 = i + j
End Function

```

Structured language support is an optional add-on module for the integrated basic compiler.

- [Using internal EEPROM memory](#)

Access to EEPROM data memory can be programmed using READ and WRITE statements. The first argument is the address of a byte in EEPROM memory and can be a constant or Byte variable. The second argument is data that is read or written (for READ statement it must be a Byte variable). It is suggested to keep interrupts disabled during the execution of WRITE statement.

```
Dim x As Byte
Dim y As Byte
x = 10
Read x, y
Write 11, y
```

- [Using internal A/D converter module](#)

ADCIN statement is available as a support for internal A/D converter. Its first argument is ADC channel number and the second argument is a variable that will be used to store the result of A/D conversion. ADCIN statement uses two parameters ADC_CLOCK and ADC_SAMPLEUS that have default values 3 and 20. These default values can be changed using DEFINE directive. ADC_CLOCK parameter determines the choice for ADC clock source (allowed range is 0-3 or 0-7 depending on the device used). ADC_SAMPLEUS parameter sets the desired ADC acquisition time in microseconds (0-255). ADCIN statement presupposes that the corresponding pin is configured as an analog input (TRISA, ADCON1 register and on some devices ANSEL register). Here is one example:

```
Dim v(5) As Byte
Dim vm As Word
Dim i As Byte

Define ADC_CLOCK = 3
Define ADC_SAMPLEUS = 50
TRISA = 0xff
TRISB = 0
ADCON1 = 0

For i = 0 To 4
  Adcin 0, v(i)
Next i

vm = 0
For i = 0 To 4
  vm = vm + v(i)
Next i
vm = vm / 5

PORTB = vm.LB
```

- [Using interrupts](#)

Interrupt routine should be placed as all other subroutines after the END statement. It should begin with ON INTERRUPT and end with RESUME statement. If arithmetic operations, arrays or any other complex statements are used in interrupt routine, then SAVE SYSTEM statement should be placed right after ON INTERRUPT statement to save the content of registers used by system. ENABLE and DISABLE statements can be used in main program to control GIE bit in INTCON register. RESUME statement will set the GIE bit and enable new interrupts. For example:

Example 1:

```
Dim x As Byte
x = 255
TRISA = 0
PORTA = x
```

```

INTCON.INTE = 1
Enable
End

On Interrupt
    x = x - 1
    PORTA = x
    INTCN.INTF = 0
Resume

```

Example 2:

```

Dim t As Word
t = 0
TRISA = 0xff
ADCON1 = 0
TRISB = 0
OPTION_REG.T0CS = 0
INTCON.T0IE = 1
Enable
loop:
    Adcin 0, PORTB
Goto loop
End

```

```

On Interrupt
    Save System
    t = t + 1
    INTCN.T0IF = 0
Resume

```

• Serial communication using internal hardware UART

The support for both hardware and software serial communication is also available. HSEROPEN, HSEROUT, HSERIN and HSERGET statements can be used with PIC devices that have internal hardware UART. HSEROPEN statement sets up the hardware UART. Its only argument is baud rate and allowed values are: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 56000 and 57600. If the argument is omitted UART will be set up for 9600 baud rate. If parameter ALLOW_MULTIPLE_HSEROPEN is set to 1 using DEFINE directive, it will be possible to use HSEROPEN statement more than once in the program, for example to change selected baud rate. If ALLOW_ALL_BAUDRATES parameter is set to 1 using DEFINE directive all baud rates in the range 100-57600 will be allowed. HSEROUT statement is used for serial transmission. HSEROUT statement may have multiple arguments separated by ','. You can use strings, LF keyword for Line Feed character or CRLF keyword for Carriage Return - Line Feed sequence, constants and variables. If '#' sign is used before the name of a variable then its decimal representation is sent to the serial port. HSERIN statement can be used to load a list of Byte and Word variables with the values received on serial port. This statement will wait until the required number of bytes is received on serial port. HSERGET statement have one argument that must be a Byte variable. If there is a character waiting in the receive buffer it will be loaded in the variable, otherwise 0 value will be loaded. Here are some examples:

Example 1:

```

Dim i As Byte
Hseropen 38400
waitMs 1000
For i = 20 To 0 Step -1
    Hserout "Number: ", #i, CrLf
    waitMs 500
Next i

```

Example 2:

```

Dim i As Byte
Hseropen 19200

```

```

loop:
  Hserin i
  Hserout "Number: ", #i, CrLf
  Goto loop

```

Example 3:

```

Dim i As Byte
Hseropen 19200
loop:
  Hserget i
  If i > 0 Then
    Hserout "Number: ", #i, CrLf
    WaitMs 50
  Endif
  Goto loop

```

• Software UART implementation

On all supported PIC devices software serial communication can be implemented with SEROUT and SERIN statements. The first argument of both statements must be one of the microcontroller's pins, and the second argument is baud rate: 300, 600, 1200, 2400, 4800, 9600 or 19200. Using higher baud rates with low clock frequency could cause framing errors. For SEROUT statement then follows the list of arguments to be sent to serial port. You can use strings, LF keyword for Line Feed character or CRLF keyword for Carriage Return - Line Feed sequence, constants and variables. If '#' sign is used before the name of a variable then its decimal representation is sent to the serial port. SEROUT statement uses SEROUT_DELAYUS parameter that can be set by DEFINE directive and has default value of 1000 microseconds. This defines the delay interval before a character is actually sent to the port and it is used to increase the reliability of software SEROUT routine. For SERIN statement then follows the list of Byte and Word variables to be loaded with the values received on serial port. This statement will wait until the required number of bytes is received on serial port. For serial interface with inverted logic levels there are SERININV and SEROUTINV statements available. Some examples:

Example 1:

```

Define SEROUT_DELAYUS = 5000
Serout PORTC.6, 1200, "Hello world!", CrLf

```

Example 2:

```

Dim i As Byte
loop:
  Serin PORTC.7, 9600, i
  Serout PORTC.6, 9600, "Number: ", #i, CrLf
  Goto loop

```

• Interfacing character LCDs

Basic compiler also features the support for LCD modules based on HD44780 or compatible controller chip. Prior to using LCD related statements, user should set up LCD interface using DEFINE directives. Here is the list of available parameters:

LCD_BITS - defines the number of data interface lines (allowed values are 4 and 8; default is 4)

LCD_DREG - defines the port where data lines are connected to (default is PORTB)

LCD_DBIT - defines the position of data lines for 4-bit interface (0 or 4; default is 4), ignored for 8-bit interface

LCD_RSREG - defines the port where RS line is connected to (default is PORTB)

LCD_RSBIT - defines the pin where RS line is connected to (default is 3)

LCD_EREG - defines the port where E line is connected to (default is PORTB)

LCD_EBIT - defines the pin where E line is connected to (default is 2)

LCD_RWREG - defines the port where R/W line is connected to (set to 0 if not used; 0 is default)

LCD_RWBIT - defines the pin where R/W line is connected to (set to 0 if not used; 0 is default)

LCD_COMMANDUS - defines the delay after LCDCMDOUT statement (default value is 5000)

LCD_DATAUS - defines the delay after LCDOUT statement (default value is 100)

LCD_INITMS - defines the delay for LCDINIT statement (default value is 100)

The last three parameters should be set to low values when using integrated LCD module simulator. If R/W line is connected to microcontroller and parameter LCD_READ_BUSY_FLAG is set to 1 using DEFINE directive, then these delay parameters will be ignored by compiler and correct timing will be implemented by reading the status of the busy flag in the LCD.

LCDINIT statement should be placed in the program before any of LCDOUT (used for sending data) and LCDCMDOUT (used for sending commands) statements. Numeric constant argument of LCDINIT is used to define the cursor type: 0 = no cursor (default), 1 = blink, 2 = underline, 3 = blink + underline. LCDOUT and LCDCMDOUT statements may have multiple arguments separated by ','. Strings, constants and variables can be used as arguments of LCDOUT statement. If '#' sign is used before the name of a variable then its decimal representation is sent to the LCD module. Constants and variables can be used as arguments of LCDCMDOUT statement and the following keywords are also available: LcdClear, LcdHome, LcdLine2Home, LcdDisplayOn, LcdDisplayOff, LcdCurOff, LcdCurBlink, LcdCurUnderline, LcdCurBlinkUnderline, LcdLeft, LcdRight, LcdShiftLeft, LcdShiftRight, LcdLine1Clear, LcdLine2Clear, LcdLine1Pos() and LcdLine2Pos(). Argument of LcdLine1Pos() and LcdLine2Pos() can be a number in the range (1-40) or Byte data type variable. The value contained in that variable should be in the same range. LcdDisplayOn and LcdDisplayOff will turn the cursor off. Cursor related symbolic commands can be used as arguments of LCDINIT. Here are some examples:

Example 1:

```
Define LCD_BITS = 8
Define LCD_DREG = PORTB
Define LCD_DBIT = 0
Define LCD_RSREG = PORTD
Define LCD_RSBIT = 1
Define LCD_EREG = PORTD
Define LCD_EBIT = 3
Define LCD_RWREG = PORTD
Define LCD_RWBIT = 2

Lcdinit LcdCurBlink
loop:
  Lcdout "Hello world!"
  waitMs 1000
  Lcdcmdout LcdClear
  waitMs 1000
  Goto loop
```

Example 2:

```
Define LCD_BITS = 8
Define LCD_DREG = PORTB
Define LCD_DBIT = 0
Define LCD_RSREG = PORTD
Define LCD_RSBIT = 1
Define LCD_EREG = PORTD
Define LCD_EBIT = 3
Define LCD_RWREG = PORTD
Define LCD_RWBIT = 2

Dim x As Word
x = 65535
Lcdinit 3
waitMs 1000
loop:
  Lcdout "I am counting!"
  Lcdcmdout LcdLine2Home
  Lcdout #x
  x = x - 1
  waitMs 250
  Lcdcmdout LcdClear
```

Goto loop

LCD related statements will take control over TRIS registers connected with pins used for LCD interface, but if you use PORTA or PORTE pins on devices with A/D Converter Module then you should take control over the ADCON1 register to set used pins as digital I/O.

You can setup up to eight user defined characters to be used on LCD. This can easily be done with LCDDEFCHAR statement. The first argument of this statement is char number and must be in the range 0-7. Next 8 arguments form 8-line char pattern (from the top to the bottom) and must be in the range 0-31 (5-bits wide). These 8 user characters are assigned to char codes 0-7 and 8-15 and can be displayed using LCDOUT statement. After LCDDEFCHAR statement the cursor will be in HOME position. For example:

```
Lcddefchar 0, 10, 10, 10, 10, 10, 10, 10, 10
Lcddefchar 1, %11111, %10101, %10101, %10101, %10101, %10101, %10101,
%11111
Lcdout 0, 1, "Hello!", 1, 0
```

For LCDs with four lines of characters additional symbolic arguments of LCDCMDOUT statement can be used: LcdLine3Home, LcdLine4Home, LcdLine3Clear, LcdLine4Clear, LcdLine3Pos() and LcdLine4Pos(). Argument of LcdLine3Pos() and LcdLine4Pos() can be a number in the range (1-40) or Byte data type variable. The value contained in that variable should be in the same range. Prior to using these language elements, correct values determining LCD type should be assigned to LCD_LINES and LCD_CHARS parameters using DEFINE directives.

```
Define LCD_LINES = 4
Define LCD_CHARS = 16
Define LCD_BITS = 8
Define LCD_DREG = PORTB
Define LCD_DBIT = 0
Define LCD_RSREG = PORTD
Define LCD_RSBIT = 1
Define LCD_EREG = PORTD
Define LCD_EBIT = 3
Define LCD_RWREG = PORTD
Define LCD_RWBIT = 2
```

```
Lcdinit 3
loop:
  Lcdcmdout LcdClear
  Lcdcmdout LcdLine1Home
  Lcdout "This is line 1"
  Lcdcmdout LcdLine2Home
  Lcdout "This is line 2"
  Lcdcmdout LcdLine3Home
  Lcdout "This is line 3"
  Lcdcmdout LcdLine4Home
  Lcdout "This is line 4"
  waitMs 1000
  Lcdcmdout LcdLine1Clear
  Lcdcmdout LcdLine2Clear
  Lcdcmdout LcdLine3Clear
  Lcdcmdout LcdLine4Clear
  Lcdcmdout LcdLine1Pos(1)
  Lcdout "Line 1"
  Lcdcmdout LcdLine2Pos(2)
  Lcdout "Line 2"
  Lcdcmdout LcdLine3Pos(3)
  Lcdout "Line 3"
  Lcdcmdout LcdLine4Pos(4)
  Lcdout "Line 4"
  waitMs 1000
Goto loop
```

- [I2C communication with external I2C devices](#)

I2C communication can be implemented in basic programs using I2CWRITE and I2CREAD statements. The first argument of both statements must be one of the microcontroller's pins that is connected to the SDA line of the external I2C device. The second argument of both statements must be one of the microcontroller's pins that is connected to the SCL line. The third argument of both statements must be a constant value or Byte variable called 'slave address'. Its format is described in the datasheet of the used device. For example, for EEPROMs from 24C family (with device address inputs connected to ground) the value 0xA0 should be used for slave address parameter. Both statements will take control over bit 0 of slave address during communication. The fourth argument of both statements must be a Byte or Word variable (this depends on the device used) that contains the address of the location that will be accessed. If a constant value is used for address parameter it must be in Byte value range. The last (fifth) argument of I2CWRITE statement is a Byte constant or variable that will be written to the specified address, and for I2CREAD statement it must be a Byte variable to store the value that will be read from the specified address. It is allowed to use more than one 'data' argument. For I2C devices that do not support data address argument there is short form of I2C statements (I2CWRITE1 and I2CREAD1) available where slave address argument is followed with one or more data arguments directly. For some I2C slave devices it is necessary to make a delay to make sure device is ready to respond to I2CREAD statement. For that purpose there is I2CREAD_DELAYUS parameter that can be set by DEFINE directive and has default value of 0 microseconds. Also, for slower I2C devices, it might be necessary to use longer clock pulses. That can be done by setting I2CCLOCK_STRETCH parameter using DEFINE directive. This parameter will set clock stretch factor. Its default value is 1. Here is one combined example with LCD module and 24C64 EEPROM (SDA connected to RC2; SCL connected to RC3):

Example 1:

```
Define LCD_BITS = 8
Define LCD_DREG = PORTB
Define LCD_DBIT = 0
Define LCD_RSREG = PORTD
Define LCD_RSBIT = 1
Define LCD_EREG = PORTD
Define LCD_EBIT = 3
Define LCD_RWREG = PORTD
Define LCD_RWBIT = 2

Dim addr As Word
Dim data As Byte
Symbol sda = PORTC.2
Symbol scl = PORTC.3
Lcdinit 3
WaitMs 1000

For addr = 0 To 31
  Lcdcmdout LcdClear
  data = 255 - addr
  I2Cwrite sda, scl, 0xA0, addr, data
  Lcdout "Write To EEPROM"
  Lcdcmdout LcdLine2Home
  Lcdout "(", #addr, ") = ", #data
  WaitMs 1000
Next addr

For addr = 0 To 31
  Lcdcmdout LcdClear
  I2Cread sda, scl, 0xA0, addr, data
  Lcdout "Read From EEPROM"
  Lcdcmdout LcdLine2Home
  Lcdout "(", #addr, ") = ", #data
  WaitMs 1000
Next addr
```


There is a set of low-level I2C communication statements available, if it is needed to have more control over I2C communication process. I2CPREPARE statement has two arguments that must be one of the microcontroller's pins. The first argument defines SDA line and second argument defines SCL line. This statement will prepare these lines for I2C communication. I2CSTART statement will generate start condition, and I2CSTOP statement will generate stop condition. One byte can be sent to the I2C slave using I2CSEND statement. After the statement is executed C bit in STATUS register will hold the copy of the state on the SDA line during the acknowledge cycle. There are two statements that can be used to receive one byte from I2C slave. I2CRECA or I2CRECEIVEACK will generate acknowledge signal during acknowledge cycle after the byte is received. I2CRECN or I2CRECEIVENACK will generate not acknowledge signal during acknowledge cycle after the byte is received. One example:

Example 2:

```
Dim addr As Word
Dim data(31) As Byte

Symbol sda = PORTC.2
Symbol scl = PORTC.3
addr = 0

I2CPrepere sda, scl
I2Cstart
I2CSend 0xa0
I2CSend addr.HB
I2CSend addr.LB
I2Cstop
I2Cstart
I2CSend 0xa1
For addr = 0 To 30
    I2CReceiveAck data(addr)
Next addr
I2CRecN data(31)
I2Cstop
```

• Support for Serial Peripheral Interface (SPI) communication

Prior to using SPI related statements, SPI interface should be set up using DEFINE directives. There are eight available parameters to define the connection of SCK, SDI, SDO and (optionally) CS lines:

SPI_SCK_REG - defines the port where SCK line is connected to
SPI_SCK_BIT - defines the pin where SCK line is connected to
SPI_SDI_REG - defines the port where SDI line is connected to
SPI_SDI_BIT - defines the pin where SDI line is connected to
SPI_SDO_REG - defines the port where SDO line is connected to
SPI_SDO_BIT - defines the pin where SDO line is connected to
SPI_CS_REG - defines the port where CS line is connected to
SPI_CS_BIT - defines the pin where CS line is connected to

The assumed settings are active-high for Clock line and active-low for ChipSelect line. That can be changed by assigning the value 1 to SPICLOCK_INVERT and/or SPICS_INVERT parameters by DEFINE directive. For slower SPI devices, it might be necessary to use longer clock pulses. The default clock stretch factor (1) can be changed by setting SPICLOCK_STRETCH parameter.

SPIPREPARE statement (no arguments) will prepare interface lines for SPI communication. SPICSON and SPICSOFF statements will enable/ disable the ChipSelect line of the interface. One byte can be sent to the SPI peripheral using SPISEND statement. To receive a byte from the peripheral SPIRECEIVE statement should be used. To send the specified number of bits there is SPISENDBITS statement available. Its first argument should be the number of bits to be sent [1-8] and the second argument is a byte variable or constant. Here is one example for using 25C040 SPI eeprom:

Example 1:

AllDigital

```
Define SPI_CS_REG = PORTC
Define SPI_CS_BIT = 0
Define SPI_SCK_REG = PORTC
Define SPI_SCK_BIT = 3
Define SPI_SDI_REG = PORTC
Define SPI_SDI_BIT = 4
Define SPI_SDO_REG = PORTC
Define SPI_SDO_BIT = 5
SPIPrepare
```

```
Define LCD_BITS = 8
Define LCD_DREG = PORTD
Define LCD_DBIT = 0
Define LCD_RSREG = PORTE
Define LCD_RSBIT = 0
Define LCD_RWREG = PORTE
Define LCD_RWBIT = 1
Define LCD_EREG = PORTE
Define LCD_EBIT = 2
Define LCD_READ_BUSY_FLAG = 1
Lcdinit
```

```
Dim addr As Byte
Dim data As Byte
```

```
For addr = 0 To 10
    data = 200 - addr
    SPICSON
    SPISend 0x06
    SPICSOFF
    SPICSON
    SPISend 0x02
    SPISend addr
    SPISend data
    SPICSOFF
    Lcdcmdout LcdClear
    Lcdout "Write To EEPROM"
    Lcdcmdout LcdLine2Home
    Lcdout "(", #addr, ") = ", #data
    waitMs 500
Next addr
```

```
For addr = 0 To 10
    SPICSON
    SPISend 0x03
    SPISend addr
    SPIReceive data
    SPICSOFF
    Lcdcmdout LcdClear
    Lcdout "Read From EEPROM"
    Lcdcmdout LcdLine2Home
    Lcdout "(", #addr, ") = ", #data
    waitMs 500
Next addr
```

Here is the same example written for 93C86 Microwire EEPROM:

Example 2:

AllDigital

```

Define SPI_CS_REG = PORTC
Define SPI_CS_BIT = 0
Define SPICS_INVERT = 1
Define SPI_SCK_REG = PORTC
Define SPI_SCK_BIT = 3
Define SPI_SDI_REG = PORTC
Define SPI_SDI_BIT = 4
Define SPI_SDO_REG = PORTC
Define SPI_SDO_BIT = 5
SPIPrepare

Define LCD_BITS = 8
Define LCD_DREG = PORTD
Define LCD_DBIT = 0
Define LCD_RSREG = PORTE
Define LCD_RSBIT = 0
Define LCD_RWREG = PORTE
Define LCD_RWBIT = 1
Define LCD_EREG = PORTE
Define LCD_EBIT = 2
Define LCD_READ_BUSY_FLAG = 1
Lcdinit

Dim addr As Byte
Dim data As Byte

SPICSON
SPISendBits 6, %100110
SPISendBits 8, %00000000
SPICSOFF

For addr = 0 To 10
    data = 200 - addr
    SPICSON
    SPISendBits 6, %101000
    SPISendBits 8, addr
    SPISend data
    SPICSOFF
    SPICSON
    SPISend 0x00
    SPICSOFF
    Lcdcmdout LcdClear
    Lcdout "Write To EEPROM"
    Lcdcmdout LcdLine2Home
    Lcdout "(", #addr, ") = ", #data
    waitMs 500
Next addr

For addr = 0 To 10
    SPICSON
    SPISendBits 6, %110000
    SPISendBits 8, addr
    SPIReceive data
    SPICSOFF
    Lcdcmdout LcdClear
    Lcdout "Read From EEPROM"
    Lcdcmdout LcdLine2Home
    Lcdout "(", #addr, ") = ", #data
    waitMs 500
Next addr

```

- [Interfacing graphical LCDs with 128x64 dot matrix](#)

Interfacing graphical LCDs with dot matrix resolution 128x64 controlled by KS0108 or compatible chip is supported with the following list of Basic language elements: GLCDINIT, GLCDCLEAR, GLCDPSET, GLCDPRESET, GLCDPOSITION, GLCDWRITE, GLCDCLEAN, GLCDOUT, GLCDIN, GLCDCMDOUT. Prior to using Graphical LCDs related statements, user should set up the interface with the graphical LCD module using DEFINE directives. Here is the list of available parameters:

[GLCD_DREG](#) - defines the port where data lines are connected to (it has to be a full 8-pins port)

[GLCD_RSREG](#) - defines the port where RS line is connected to

[GLCD_RSBIT](#) - defines the pin where RS line is connected to

[GLCD_EREG](#) - defines the port where E line is connected to

[GLCD_EBIT](#) - defines the pin where E line is connected to

[GLCD_RWREG](#) - defines the port where R/W line is connected to

[GLCD_RWBIT](#) - defines the pin where R/W line is connected to

[GLCD_CS1REG](#) - defines the port where CS1 line is connected to

[GLCD_CS1BIT](#) - defines the pin where CS1 line is connected to

[GLCD_CS2REG](#) - defines the port where CS2 line is connected to

[GLCD_CS2BIT](#) - defines the pin where CS2 line is connected to

GLCDINIT statement should be placed somewhere at the beginning of the basic program before any other graphical LCD related statements are used. Graphical LCD related statements will take control over TRIS registers connected with pins used for LCD interface, but if you use pins that are setup as analog inputs at power-up on devices with A/D Converter and/or Comparator modules, you should take control over the appropriate register(s) (ADCON1, ANSEL, CMCON) to set used pins as digital I/O.

GLCDCLEAR statement will clear the whole display. It can be used with one optional constant argument in the range 0-255 that will be placed on every byte position on the display (128x64 graphical displays are internally divided in two 64x64 halves; both halves are divided in eight 64x8 horizontal pages; every page has its addressing number in the range 0-15; page in upper-left corner has number 0; page in lower-left corner has number 7; page in upper-right corner has number 8; page in lower-right corner has number 15; every page has 64 byte positions addressed with numbers in the range 0-63; every byte position has 8 bits; the uppermost bit is LSB and the lowermost bit is MSB). For example:

```
GLcdinit
loop:
  GLcdclear 0xaa
  waitms 1000
  GLcdclear 0x55
  waitms 1000
  goto loop
```

GLCDPSET and GLCDPRESET statements are used to turn on and turn off one of the dots on the graphical display. The first argument is the horizontal coordinate and it must be a byte data type variable or constant in the range 0-127. The second argument is the vertical coordinate and it must be a byte data type variable or constant in the range 0-63. The dot in the upper-left corner of the display is the origin with coordinates 0,0. For example:

```
Dim i As Byte
Dim j As Byte
GLcdinit
For i = 0 To 127
  For j = 0 To 63
    GLcdpset i, j
  Next j
Next i
```

GLCDCLEAN statement is used to clear a section of the page on the display. It has three arguments. The first argument is page address and it must be a byte data type variable or constant in the range 0-15. The second argument is the first byte position on the page that will be cleaned and it must be a byte data type variable or constant in the range 0-63. The third argument is the last byte position on the page that will be cleaned and it must be a byte data type variable or constant in

the range 0-63. If the last two arguments are omitted the whole page will be cleared. For example:

```
Dim i As Byte
GLcdinit
GLcdclear 0xff
For i = 0 To 15
    GLcdclean i
    waitMs 500
Next i
```

GLCDPOSITION statement is used to address a byte position on the display. It must be used before any of the GLCDWRITE, GLCDIN, GLCDOUT and GLCDCMDOUT statements. The first argument is page address and it must be a byte data type variable or constant in the range 0-15. The second argument is the target byte position on the page and it must be a byte data type variable or constant in the range 0-63. If the second argument is omitted, zero byte position is used.

GLCDWRITE statement is used to write text on the display. It will start writing from the current byte position on the display. It must be used carefully, because when the byte position (63) of the page is reached, the writing will continue from the byte position 0 staying on the same page. The width of every character written is 5 byte positions plus one clear byte position. After the statement is executed the current byte position will be at the end of the text written. GLCDWRITE statement may have multiple arguments separated by ','. Strings, constants and byte variables can be used as its arguments. Constants and variable values are interpreted as ASCII codes. If '#' sign is used before the name of a variable (byte or word data type) then its decimal representation is written. For example:

```
Dim i As Byte
GLcdinit
For i = 0 To 15
    GLcdposition i, 0
    GLcdwrite "Page: ", #i
    waitMs 250
Next i
```

GLCDOUT statement is used to write the value of the byte variable or constant at the current byte position on the display. The current byte position will be incremented by one. GLCDIN statement will read the value from the current byte position on the display and put it in the byte variable specified as its argument. GLCDCMDOUT statement is used to send low-level commands to the graphical LCD. Its argument can be a constant or byte data type variable. All these three statements can be used with multiple arguments separated by ','.

• [Using internal PWM modules](#)

Internal PWM modules (more precisely: PWM modes of CCP modules) are turned on using PWMON statement. This statement has two arguments. The first argument is module number and it must be a constant in the range 1-3. The second argument is used for mode selection. Internal PWM module can be used on three different output frequencies for each of four duty cycle resolutions supported by PWMON statement (10-bit, 9-bit, 8-bit and 7-bit). So, PWM module can be turned on with PWMON statement in 12 modes. Here is the list of all modes at 4MHz clock frequency (for other clock frequencies, the values should be proportionally adjusted):

[mode 1](#): 10-bit, 244Hz
[mode 2](#): 10-bit, 977Hz
[mode 3](#): 10-bit, 3906Hz
[mode 4](#): 9-bit, 488Hz
[mode 5](#): 9-bit, 1953Hz
[mode 6](#): 9-bit, 7813Hz
[mode 7](#): 8-bit, 977Hz
[mode 8](#): 8-bit, 3906Hz
[mode 9](#): 8-bit, 15625Hz
[mode 10](#): 7-bit, 1953Hz
[mode 11](#): 7-bit, 7813Hz
[mode 12](#): 7-bit, 31250Hz

The PWM module is initially started with 0 duty cycle, so the output will stay low until the duty cycle is changed. PWM module can be turned off with PWMOFF statement. It has only one argument - module number.

The duty cycle of PWM signal can be changed with PWMDUTY statement. Its first argument is module number. The second argument is duty cycle and it can be a constant in the range 0-1023 or byte or word data type variable. User must take care to use the proper value ranges for all PWM modes (0-1023 for 10-bit resolution, 0-511 for 9-bit resolution, 0-255 for 8-bit resolution and 0-127 for 7-bit resolution). Here is one example example:

```
Dim duty As Byte
PwMon 1, 9
loop:
  Adcin 0, duty
  PWMDuty 1, duty
  Goto loop
```

• Interfacing Radio Control (R/C) servos

For writing applications to interface R/C servos there are two statements available: SERVON and SERVOOUT. R/C servo is controlled by a train of pulses (15-20 pulses per second) whose length define the position of the servo arm. The valid length of pulses is in the range 1-2ms. These two statements have two arguments. The first argument of both statements is the microcontroller pin where the servo signal is received or transmitted. For SERVON statement that pin should be previously setup as an input pin and for SERVOOUT statement the pin should be setup for output. The second argument of SERVON statement must be a Byte variable where the length of the pulse will be saved. The pulses are measured in 10us units, so it is possible to measure pulses in the range 0.01-2.55ms. The value stored in the variable for normal servos should be in the range 100-200. The second argument of the SERVOOUT statement should be a Byte variable or constant that determines the length of the generated pulse. For proper operation of the target servo SERVOOUT statement should be executed 15-20 times during one second. Here is an example of the servo reverse operation:

```
Dim length As Byte
TRISB.0 = 1
TRISB.1 = 0
loop:
  ServoIn PORTB.0, length
  If length < 100 Then length = 100
  If length > 200 Then length = 200
  length = length - 100
  length = 100 - length
  length = length + 100
  ServoOut PORTB.1, length
  Goto loop
```

• Interfacing Stepper Motors

Prior to using stepper motor related statements, its connection and desired drive mode should be set up using DEFINE directives. There are eight available parameters to define the connection of A, B, C and D coils:

STEP_A_REG - defines the port where A coil is connected to
STEP_A_BIT - defines the pin where A coil is connected to
STEP_B_REG - defines the port where B coil is connected to
STEP_B_BIT - defines the pin where B coil is connected to
STEP_C_REG - defines the port where C coil is connected to
STEP_C_BIT - defines the pin where C coil is connected to
STEP_D_REG - defines the port where D coil is connected to
STEP_D_BIT - defines the pin where A coil is connected to

Coils A and C are actually parts of one single coil with common connection. The same is valid for B and D coil connections. There is also STEP_MODE parameter used to define the drive mode. If it is set to 1 (default) the motor will be driven in full-step mode. The value 2 should be used for half-step

mode. The first basic statement that should be used is STEPHOLD. It will configure used pins as outputs and also energize A and B coils to fix the rotor in its initial position. For moving rotor in clockwise and counterclockwise directions there are STEPCW and STEPCCW statements available. Their first argument is the number of rotor steps that will be performed and it can be Byte data type constant or variable. The second argument defines the delay between consecutive steps expressed in microseconds by a Byte or Word data type variable or constant. If using STEPCW statement results in rotor movement in counterclockwise direction then connection settings for B and D coils should be exchanged. Here are two examples (the second example uses delays suitable for simulation in the simulator):

Example 1:

```
AllDigital
ADCON1 = 0x0e
Define STEP_A_REG = PORTB
Define STEP_A_BIT = 7
Define STEP_B_REG = PORTB
Define STEP_B_BIT = 6
Define STEP_C_REG = PORTB
Define STEP_C_BIT = 5
Define STEP_D_REG = PORTB
Define STEP_D_BIT = 4
Define STEP_MODE = 2

waitMs 1000
StepHold
waitMs 1000

Dim an0 As Word

loop:
  Adcin 0, an0
  an0 = an0 * 60
  an0 = an0 + 2000
  StepCW 1, an0
  Goto loop
```

Example 2:

```
AllDigital
Define STEP_A_REG = PORTB
Define STEP_A_BIT = 7
Define STEP_B_REG = PORTB
Define STEP_B_BIT = 6
Define STEP_C_REG = PORTB
Define STEP_C_BIT = 5
Define STEP_D_REG = PORTB
Define STEP_D_BIT = 4
Define STEP_MODE = 2

waitUs 300
StepHold
waitUs 1000

loop:
  StepCCW 16, 300
  waitUs 1000
  StepCW 24, 300
  waitUs 1000
  Goto loop
```

• **Interfacing 1-WIRE devices**

Prior to using 1-WIRE related statements, user should define the pin where the device is connected

to using DEFINE directives. Available parameters are 1WIRE_REG and 1WIRE_BIT. For example:

```
Define 1WIRE_REG = PORTB
Define 1WIRE_BIT = 0
```

Initialization sequence can be performed by 1WIREINIT statement. It can have an optional argument (Bit data type variable) that will be set to 0 if the presence of the device has been detected and set to 1 if there is no device on the line.

Individual bits (time slots) can be sent to and received from the device using 1WIRESENDBIT and 1WIREGETBIT statements. Both statements can have multiple arguments - comma separated list of Bit data type variables (or Bit constants for 1WIRESENDBIT statement).

1WIRESENDBYTE and 1WIREGETBYTE statements can be used to send to and receive bytes from the device. Both statements can have multiple arguments - comma separated list of Byte data type variables (or Byte constants for 1WIRESENDBYTE statement). Here is one example for measuring temperature using DS18S20 device:

```
Dim finish As Bit
Dim temp As Byte
Dim sign As Byte

1wireInit
1wireSendByte 0xcc, 0x44
waitMs 1
loop:
    1wireGetBit finish
    If finish = 0 Then Goto loop
    1wireInit
    1wireSendByte 0xcc, 0xbe
    1wireGetByte temp, sign
```

This example can be very short by using two DS18S20 specific high level basic statements. DS18S20START statement will initiate a single temperature conversion. According to the device datasheet the conversion will be completed in at most 750ms. After that period the measured value can be read by DS18S20READT statement that requires two Byte data type variables as arguments. The first argument will contain the temperature value in 0.5 degrees centigrade units (for example, the value 100 represents the temperature of 50 degrees). The second argument will contain the value 0x00 if the temperature is positive and 0xFF value if it is negative. For example:

```
Dim temp As Byte
Dim sign As Byte

DS18S20Start
waitMs 1000
DS18S20ReadT temp, sign
```

• Advanced features

If STARTFROMZERO directive is used the compiler will start the program from zero flash program memory location (reset vector) and use the available program memory continuously. Interrupt routine if used should be implemented by using inline assembler code. The compiler will also leave control over PCLATH register to the user supposing that all code is placed in the same program memory page. This advanced feature can be used when developing bootloader applications, for example.