Oshonsoft PIC Basic Compiler Reference Manual. [EG]

PIC Simulator Ver 7.0 [22 March 2010] Rev: 31 Jan 2014

Index Page.

Basic Compiler Keywords. Index

Configuration Commands.
CLOCK_FREQUENCY, CONF_WORD, CONF_WORD_2

WAIT State Commands.
SIMULATION_WAITMS_VALUE, WAITMS, WAITUS

Data Type Assignments.
DIM, AS, BIT, BYTE,WORD,LONG,CONST

Symbolic Names.
SYMBOL

Compiler Directives.
ALLDIGITAL
DEFINE, RESERVE
STARTFROMZERO
ASM,END

Logical Operators.
AND, OR, XOR, NAND, NEXT, NOR, NOT, NXOR

Arithmetic Operators.
+, -, *, /, MOD, SQR

Basic Structures.
SELECT CASE, CASE, ENDSELECT
FOR, STEP, TO, WHILE, WEND
IF, THEN, ELSE, ENDIF
GOSUB, GOTO, RETURN

Compiler Labels.
WREG
CRLF,LF
LOOKUP, POINTER
HIGH,LOW
FALSE, TRUE, TOGGLE,
SHIFTLEFT, SHIFTRIGHT

Interrupt Commands.
DISABLE, ENABLE, ON INTERRUPT, SAVE SYSTEM, RESUME

Debugger Statements.
BREAK, HALT

Structured Language. [*Option*]
FUNCTION, END FUNCTION
PROC, END PROC, EXIT
CALL

PIC Internal Module Commands. Index

EEPROM, WRITE,READ,

ADCIN, ADC_CLOCK, ADC_SAMPLEUS

COUNT,COUNT_MODE,FREQOUT

PWMON, PWMDUTY, PWMOFF

SPI_CS_REG, SPI_CS_BIT, SPI_SCK_REG, SPI_SCK_BIT, SPI_SDI_REG, SPI_SDI_BIT,
SPI_SDO_REG, SPI_SDO_BIT, SPICS_INVERT, SPICLOCK_INVERT,
SPICLOCK_STRETCH,
SPICSON, SPICSOFF, SPIPREPARE, SPISEND, SPISENDBITS, SPIRECEIVE,

1WIRE_REG, 1WIRE_BIT, 1WIREINIT, 1WIRESENDBIT, 1WIREGETBIT,
1WIRESENDBYTE, 1WIREGETBYTE

ALLOW_ALL_BAUDRATES, ALLOW_MULTIPLE_HSEROPEN
HSERGET, HSERIN, HSEROUT, HSEROPEN
SERIN, SERININV, SEROUT, SEROUTINV, SEROUT_DELAYUS,

I2CWRITE, I2CREAD,I2CREAD_DELAYUS, I2CCLOCK_STRETCH, I2CWRITE1
I2CREAD1, I2CPREPARE, I2CSTART, I2CSTOP, I2CSEND, I2CRECA, I2CRECEIVEACK,
I2CRECN,I2CRECEIVENACK

PIC External Module Commands.

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, LCD_LINES, LCD_CHARS, LCDINIT, LCDOUT,
LCDCMDOUT, LCDCLEAR, LCDHOME, LCDDISPLAYON, LCDDISPLAYOFF,
LCDCUROFF,
LCDCURBLINK, LCDCURUNDERLINE, LCDCURBLINKUNDERLINE, LCDLEFT,
LCDRIGHT,LCDSHIFTLEFT, LCDSHIFTRIGHT, LCDLINE1HOME, LCDLINE2HOME,
LCDLINE3HOME, LCDLINE4HOME, LCDLINE1CLEAR, LCDLINE2CLEAR,
LCDLINE3CLEAR, LCDLINE4CLEAR,
LCDLINE1POS, LCDLINE2POS, LCDLINE3POS, LCDLINE4POS, LCDDEFCHAR,

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,

DS18S20START, DS18S20READT

SERVOIN, SERVOOUT,STEP, 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,

Standard Basic Language Elements. Index

Default extension for basic source files is BAS.
The compiler output is assembler source file (with ASM extension) that can be translated to binary code using integrated assembler.
Smart editor marks all reserved keywords in different color, that simplifies debugging process.
BASIC compiler's assembler output has all necessary comment lines, that makes it very useful for educational purposes, also.

Configuration Parameters.
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).

<u>These parameters should be setup at the beginning of the basic Program.</u>
For example:

```
DEFINE CONF_WORD = 0x3F72
DEFINE CLOCK_FREQUENCY = 20
```

Data Types.
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) [Option}

Declarations.
Declarations may be placed anywhere in the program.
All variables are considered global.
The total number of variables is limited by the available microcontroller RAM memory.
Variables are declared using DIM statement:

```
DIM A AS BIT
DIM B AS BYTE
DIM X AS WORD
DIM Y AS LONG
```

If necessary, variable address can be specified during declaration: DIM X AS BYTE @ 0x050

Arrays.
It is also possible to use one-dimensional arrays. For example: DIM A(10) AS BYTE declares an array of 10 Byte variables with array index in the range [0-9].

Constants.
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).
Constants can also be assigned to symbolic names using CONST directive:

```
DIM A AS WORD
CONST PI = 314
A = PI
```

Arithmetic Operators. Index

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 A AS WORD
DIM B AS WORD
DIM X AS WORD
A = 123
B = A * 234
X = 2
X = (12345 - B * X) / (A + B)
```

Square root of a number (0-65535 range) can be calculated using SQR function:
DIM A AS WORD A = 3600 A = SQR(A)

Logical Operators.
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:

```
DIM A AS BIT
DIM B AS BIT
DIM X AS BIT
X = NOT A
X = A AND B
X = A OR B
X = A XOR B
X = A NAND B
X = A NOR B
X = A NXOR B
```

```
DIM A AS WORD
DIM B AS WORD
A = A OR B
PORTB = PORTC AND %11110000
```

TRUE and FALSE.
Keywords True and False are also available for Bit type constants.

For example:

```
DIM A AS BIT
DIM B AS BYTE
A = TRUE
B = 0x55
B = %01010101
```

SYMBOLIC Names. Index

It is also possible to use symbolic names (symbols) in programs:

```
SYMBOL LED1 = PORTB.0
LED1 = 1
SYMBOL AD_ACTION = ADCON0.GO_DONE
```

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
    ASM:    MOVLW 0xFF
    ASM:    MOVWF VARNAME
```

When working with inline assembler code, it could be useful to use the 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
```

BASIC Structures.

Four standard BASIC structures are supported:

FOR-TO-STEP-NEXT
WHILE-WEND
IF-THEN-ELSE-ENDIF
SELECT CASE-CASE-ENDSELECT

Here are several examples:

```
DIM A AS BYTE
TRISB = 0
A = 255
WHILE A > 0
PORTB = A
A = A - 1
WAITMS 100
WEND
PORTB = A
TRISB = 0
loop:
IF PORTA.0 THEN
PORTB.0 = 1
ELSE
PORTB.0 = 0
ENDIF
GOTO loop
DIM A AS WORD
TRISB = 0
FOR A = 0 TO 10000 STEP 10
PORTB = A.LB
NEXT A
```

[Index](#)

```
DIM A AS BYTE
DIM B AS BYTE
DIM X AS BYTE
B = 255
X = 2
TRISB = 0
FOR A = B TO 0 STEP -X
PORTB = A
NEXT A
```

```
DIM A AS BYTE
loop:
SELECT CASE A
CASE 255
A = 1
CASE <= 127
A = A + 1
CASE ELSE
A = 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.

Standard Short Form Labels.

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 A AS BYTE
DIM B AS WORD
A = B.HB
A = B.LB 'This statement is equivalent to A = B
B.HB = A
B.LB = A
```

B = A 'This statement will also clear the high byte of B 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 A AS BYTE
DIM B AS WORD
DIM X AS LONG
A = X.LB
B = X.HW
```

Short Form cont ..... Index

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 A AS BIT
DIM B AS BYTE
A = B.7
B.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
```

Structured 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
```

POINTER Command. Index

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
```

LOOKUP Command.

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 loop:
TRISB = %00000000
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
```

BIT Control.

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
```

COUNT Command. Index

If it is necessary to count the number of pulses that come to one of the micrcontroller'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
```

FREQUENCY Command.

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
```

SHIFT Commands. Index

SHIFTLEFT 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:

```
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
```

```
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
```

Assembler Coding.

Lines of assembler source code may be placed anywhere in basic source program and must begin with the 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
    ASM:       MOVLW 0xFF
    ASM:       MOVWF VARNAME
```

When working with inline assembler code, it could be useful to use working register , W 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
```

Interrupts. Index

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 the GIE bit in INTCON register. [global intr enable]

RESUME statement will set the GIE bit and enable new interrupts.
For example:

```
DIM A AS BYTE
A = 255
TRISA = 0
PORTA = A
INTCON.INTE = 1
ENABLE
END
ON INTERRUPT
A = A - 1
PORTA = A
INTCON.INTF = 0
RESUME
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
INTCON.T0IF = 0
RESUME
```

Internal EEPROM Memory. Index

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
```

The GOTO statement uses line label name as argument. Line labels must be followed by colon mark ":".
Here is one example:

```
DIM A AS WORD
A = 0
loop: A = A + 1
GOTO loop
```

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 A AS BYTE
DIM B AS BYTE
A = 10
READ A, B
WRITE 11, B
```

Internal Analog to Digital Module. Index

The **ALLDIGITAL** statement can be used at the beginning of the basic program.
To setup all pins for digital purposes.

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.

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 (TRIS, 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
```

Internal Hardware UART Communications. Index

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:

```
DIM I AS BYTE
HSEROPEN 38400
WAITMS 1000
FOR I = 20 TO 0 STEP -1
HSEROUT "Number: ", #I, CrLf
WAITMS 500
NEXT I
```

```
DIM I AS BYTE
HSEROPEN 19200
loop:
HSERIN I
HSEROUT "Number: ", #I, CrLf
GOTO loop
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 . Index

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:

```
DEFINE SEROUT_DELAYUS = 5000
SEROUT PORTC.6, 1200, "Hello world!", CrLf
DIM I AS BYTE loop:
SERIN PORTC.7, 9600, I
SEROUT PORTC.6, 9600, "Number: ", #I, CrLf
GOTO loop
```

External I2C devices Communications. Index

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 forth 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.

External I2C devices cont........ Index

Here is one combined example with LCD module and 24C64 EEPROM
(SDA connected to RC2; SCL connected to RC3):

```
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
```

low-level I2C communication . Index

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 acknowlegde cycle after the byte is received.

I2CRECN or I2CRECEIVENACK will generate not acknowledge signal during acknowlegde cycle after the byte is received.

One example:
```
DIM ADDR AS WORD
DIM DATA(31) AS BYTE
SYMBOL SDA = PORTC.4
SYMBOL SCL = PORTC.3
ADDR = 0
I2CPREPARE 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
```

Serial Peripheral Interface (SPI) communication . Index

Prior to using SPI related statements, SPI interface should 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.

(SPI) communication cont .....

Here is one example for using 25C040 SPI eeprom: Index

```
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 = 5
Define SPI_SDO_REG = PORTC
Define SPI_SDO_BIT = 4
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
```

93C86 Microwire EEPROM Index

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

```
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 = 5
Define SPI_SDO_REG = PORTC
Define SPI_SDO_BIT = 4
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

Alpha-Numeric LCDs [HD44780 or compatible controller device]. Index

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.

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.

Here is the list of available parameters:

```
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
```

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 the 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. Index

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.

Alpha-Numeric LCDs cont.....

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:
```
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
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 A AS WORD
A = 65535 Index
LCDINIT 3
WAITMS 1000
loop:
```

Alpha-Numeric LCDs cont.....

```
LCDOUT "I am counting!"
LCDCMDOUT LcdLine2Home
LCDOUT #A
A = A - 1
WAITMS 250
LCDCMDOUT LcdClear
GOTO loop
```

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,
```

Four Line LCD's. Index

For LCDs with four lines of characters additional symbolic arguments of
the 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
```

Interfacing graphical LCD’s Index

Interfacing graphical LCDs with dot matrix resolution 128x64 controlled by KS0107 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
this statement should be placed somewhere at the beginning of the basic program before any other graphical LCD related stetements 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 internaly 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
```

Graphic LCD's cont.......... Index

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.

Graphic LCD's cont........... Index

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 Index

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. Index

For writing applications to interface R/C servos there are two statements available SERVOIN 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 SERVOIN 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 SERVOIN 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 1-WIRE devices . Index

Prior to using 1-WIRE related statements, the 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
```

Interfacing Stepper Motors Index

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.

Stepper Motors cont......... Index

Here are two examples (the second example uses delays suitable for simulation in the simulator):

```
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
```

```
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
```

Debug. Index

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

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.

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.

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

WAIT Delays. Index

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 A AS WORD
A = 100
WAITMS A
WAITUS 50
```

PLEASE 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

Example. **Define SIMULATION_WAITMS_VALUE = 1**

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.

Advanced features Index

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.

Structered Language Support. [Option]

Structured language support (procedures and functions) - ***optional module***
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 A AS BYTE
FOR A = 0 TO 255
CALL portb_display(A)
WAITMS 100
NEXT A
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 A AS BYTE
DIM B AS WORD
FOR A = 0 TO 255
B = square(A)
NEXT A
END

FUNCTION square(arg1 AS WORD) AS WORD
square = arg1 * arg1
END FUNCTION
```

The list of all Basic compiler keywords. Index

1WIRE_REG, 1WIRE_BIT, 1WIREINIT, 1WIRESENDBIT, 1WIREGETBIT, 1WIRESENDBYTE, 1WIREGETBYTE

ADCIN, ADC_CLOCK, ADC_SAMPLEUS,ALLDIGITAL,ALLOW_ALL_BAUDRATES, ALLOW_MULTIPLE_HSEROPEN, AND, AS, ASM

BIT, BREAK, BYTE

CALL, CASE, CLOCK_FREQUENCY, CONF_WORD, CONF_WORD_2, CONST, COUNT, COUNT_MODE, CRLF,

DEFINE, DIM, DISABLE, DS18S20START, DS18S20READT

EEPROM, ELSE, ENABLE, END, END FUNCTION, END PROC, ENDIF, ENDSELECT, EXIT

FALSE, FOR, FREQOUT, FUNCTION

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,
GOSUB, GOTO

HALT, HIGH, HSERGET, HSERIN, HSEROUT, HSEROPEN

I2CWRITE, I2CREAD, I2CREAD_DELAYUS, I2CCLOCK_STRETCH, I2CWRITE1, I2CREAD1, I2CPREPARE, I2CSTART, I2CSTOP, I2CSEND, I2CRECA, I2CRECEIVEACK, I2CRECN, I2CRECEIVENACK,

IF

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, LCD_LINES, LCD_CHARS, LCDINIT, LCDOUT, LCDCMDOUT, LCDCLEAR, LCDHOME, LCDDISPLAYON, LCDDISPLAYOFF, LCDCUROFF, LCDCURBLINK, LCDCURUNDERLINE, LCDCURBLINKUNDERLINE, LCDLEFT, LCDRIGHT, LCDSHIFTLEFT, LCDSHIFTRIGHT, LCDLINE1HOME, LCDLINE2HOME, LCDLINE3HOME, LCDLINE4HOME, LCDLINE1CLEAR, LCDLINE2CLEAR, LCDLINE3CLEAR,LCDLINE4CLEAR, LCDLINE1POS, LCDLINE2POS, LCDLINE3POS, LCDLINE4POS, LCDDEFCHAR,
LF, LONG,
LOOKUP,
LOW,

MOD

NAND, NEXT, NOR, NOT, NXOR

ON INTERRUPT, OR

POINTER, PROC, PWMON, PWMDUTY, PWMOFF

READ, RESERVE, RESUME, RETURN

SAVE SYSTEM, SELECT CASE, SERIN, SERININV, SEROUT, SEROUTINV, SEROUT_DELAYUS, SERVOIN, SERVOOUT, SHIFTLEFT, SHIFTRIGHT, SIMULATION_WAITMS_VALUE, SPI_CS_REG, SPI_CS_BIT,SPI_SCK_REG, SPI_SCK_BIT, SPI_SDI_REG, SPI_SDI_BIT, SPI_SDO_REG, SPI_SDO_BIT,SPICS_INVERT, SPICLOCK_INVERT, SPICLOCK_STRETCH, SPICSON, SPICSOFF, SPIPREPARE,SPISEND, SPISENDBITS, SPIRECEIVE, SQR, STARTFROMZERO, STEP, 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, SYMBOL

THEN, TO, TOGGLE, TRUE

WAITMS, WAITUS, WEND, WHILE, WORD

WREG, WRITE, XOR.