MadLab PIC BASIC Reference

Written by James Hutchby

info@madlab.org
www.madlab.org

Contents

BASIC Programs

A BASIC program consists of a number of lines (statements) containing labels, instructions, constants, and variables.

Labels identify sections of a program. They are up to 64 characters long, and can contain letters, numbers and the underscore character _, but the first character must be a letter. The definition of a label requires a colon immediately following the label, but the colon is not needed when the label is used in a **GOTO** or **GOSUB** instruction. Labels are not case-sensitive. See **GOTO** for an example of the use of a label.

Instructions can be in upper or lower case. Instructions do not have to be indented in a line but it is good practice to do so. The keywords for instructions cannot be used as labels or variable names.

Constants can be decimal (base 10), hexadecimal (base 16), binary (base 2) or character. Hexadecimal constants are preceded with a $ symbol, binary constants with a % symbol, and characters are enclosed in double quotes. Embedded double quotes in characters and strings are preceded by a backslash character. See **LET** for examples of constants.

Variables can be single-bit, 8-bit, 16-bit, signed or unsigned. The names of variables follow the same rules as labels and are not case-sensitive. See **DIM**.

All arithmetic is integer arithmetic. Overflows are not detected.

Complex expressions are allowed with operator precedence.

The installed "examples" directory contains sample BASIC programs.

Execution starts at the first statement in a program and continues until the last statement.

BASIC Instructions

'
ANALOG <pin>{,**0**|**1**|**2**|**4**}
CLEAR <pin>|<variable>
CLS LCD|USB
DATA <address>,<constant>|<string>{,<constant>|<string>,...}
DIM <variable>{**:1**|**1u**|**8**|**8u**|**8s**|**16**|**16u**|**16s**|**u**|**s**}{,<variable>,...}
EEPROM <address>,<constant>|<string>{,<constant>|<string>,...}
END
FOR <variable> = <start> **TO** <end> {**STEP** <step>}
GOSUB <label>
GOTO <label>
HIGH <pin>|<variable>
IF <expression> **THEN** <label>|{**THEN**} <instruction> {**ELSE** <label>|<instruction>}
INPUT <pin>{,**off**|**pullup**}
LCD <pin_d7>,<pin_d6>,<pin_d5>,<pin_d4>,<pin_rs>,<pin_rw>,<pin_e>,<rows>,<cols>
{**LET**} <variable> = <expression>
LOW <pin>|<variable>
MOTORS <pin1A>,<pin1B>,<pin2A>,<pin2B>{,**off**|**on**}
NEXT {<variable>}
ON <expression> **GOTO**|**GOSUB** <label>{,<label>,...}
OUTPUT <pin>
PAUSE <duration>
PRINT LCD|SPI|UART|USB <expression>|<string>{,|;<expression>|<string>,|;...}
PWM <pin>,<period>,<duty>
RANDOM {<variable>}
READ <address>,<variable>{,<variable>,...}
RECEIVE SPI|UART|USB <variable>{,<variable>,...}
REM
RESET <pin>|<variable>
RETURN
SEND LCD|SPI|UART|USB <expression>|<string>{,<expression>|<string>,...}
SERVO <pin>{,**off**|**on**}
SET <pin>|<variable>
SPI <clock>,<mode>
STEPPER <pin1A>,<pin1B>,<pin2A>,<pin2B>,<speed>{,**off**|**standard**|**high_torque**|**half_step**}
SYMBOL <symbol> = <constant>|<variable>
TOGGLE <pin>|<variable>
UART <baud>,<data>,<parity>,<stop>
WHEN <pin> = **0**|**1 GOSUB** <label>
WHEN <timer> = <period> **GOSUB** <label>
WRITE <address>,<expression>|<string>{,<expression>|<string>,...}

| indicates alternatives, {} indicates optional

<symbol>, <variable>, <label> = string of letters, numbers and underscores with a letter as the first character
<expression> = expression involving constants, symbols, variables and operators
<constant> = simple constant or expression that evaluates to a constant
<string> = string of characters delimited by double quotes
<pin> = 1 to 12, or symbolic pin name

'

Indicates a comment (remark). Comments are aids to the understanding of a program and are ignored by the compiler. **'** can be used by itself on a line, or after other instructions.

See also **REM**.

```
' this is a comment
        IF x > 10 GOTO too_big    ' test x against upper limit
```

ANALOG <pin>{,**0**|**1**|**2**|**4**}

Configures a pin as an analog input. Pins 1, 2, 3, 5, 6, 8 and 9 can be analog inputs.

The second argument specifies the voltage range. Four options are available (0, 1, 2 and 4 respectively): 0V to the processor supply voltage Vdd (normally 5V), 0 to 1.024V, 0 to 2.048V, and 0 to 4.096V. The latter three are precision voltage references. The default is Vdd.

Note that the voltage range applies to all analog inputs equally. It is not possible to have different ranges simultaneously on different analog pins.

The analog voltage level on a pin is read via the variables *analog1*, *analog2* etc. These variables are unsigned 16-bit variables with a range from 0 to 1023 (10-bit resolution) corresponding to the specified voltage range.

Be careful not to apply voltages greater than 5V to these pins otherwise you might damage the PIC. Note that the maximum recommended impedance for analog sources is 10kΩ.

```
ANALOG pin1                     ' pin 1 analog input
LET x = analog1                 ' analog voltage on pin 1
IF x > 512 THEN ...             ' if voltage greater than 2.5V ...

ANALOG 2,1                      ' pin 2 analog input, 0 to 1.024V
IF analog2 < 100 THEN ...       ' if voltage less than 100mV ...
```

CLEAR <pin>|<variable>

See **LOW**.

CLS LCD|USB

Clears the LCD display or the Console window (if open).

The Console window can also be cleared by clicking on the Clear button in its menu bar.

```
CLS LCD                          ' clear LCD display, home cursor
CLS USB                          ' clear Console window
PRINT USB "The result is ",result
CLS                              ' also clears Console window
```

DATA <address>,<constant>|<string>{,<constant>|<string>,...}

Initialises EEPROM data memory with 8-bit data. EEPROM memory can be used to set up data tables, or as additional memory for variables etc. **DATA** statements can appear anywhere in a program.

<address> is a constant or constant expression that evaluates to an address in the range 0 to 239. An address outside this range causes an error. The data to be stored can be a constant expression or a string. Data is stored in consecutive memory locations, and strings are stored one byte per character without a zero terminator.

EEPROM is a synonym for this instruction. See also **READ**, **WRITE** (this instruction differs from **WRITE** in that the data is written when the program is downloaded, not when it is executed).

```
DATA 10,40,45,50,55,60          ' data table with 5 entries
READ 10+n,x                     ' reading the n'th entry

DATA 20,"an error message"

DATA 30,1234>>8,1234&$ff        ' storing a 16-bit constant
```

DIM <variable>{**:1**|**1u**|**8**|**8u**|**8s**|**16**|**16u**|**16s**|**u**|**s**}{,<variable>,...}

Defines a variable and optionally specifies its size and signed properties. Variables do not have to be defined to be used but it is good practice to do so. Variables can then be made no larger than they need to be. For example, a flag variable which only holds the values 0 and 1 can be efficiently stored in a single bit rather than in a complete byte. Memory for variables is in reasonably short supply so it is advisable not to waste it. 16-bit variables should be used sparingly because the compiler needs to generate extra code to handle them which means that program memory fills more quickly. Additionally 16-bit code runs more slowly. Signed variables also introduce an overhead compared to unsigned variables.

DIM if used must precede any reference to the variable. It is usual to group all the **DIM** statements at the beginning of a program. If a variable is used without (or before) being defined then its properties default to those specified in the Compiler Options (but see **HIGH**, **LOW**, **TOGGLE**).

The optional modifiers specify the size and signed properties of the variable:

:1	single-bit variable, 0 or 1
:8u	8-bit unsigned variable, in the range 0 to 255
:8s	8-bit signed variable, in the range –128 to 127
:16u	16-bit unsigned variable, in the range 0 to 65535
:16s	16-bit signed variable, in the range –32768 to 32767

Single-bit variables are always unsigned.

It is possible to force the use of DIM statements by checking the 'DIM variables before use' option in Compiler Options. As well as being good programming discipline, this also catches any mis-spellings in variable names.

Note that all variables are cleared at the start of a program.

```
DIM x:16u                ' 0 to 65535
DIM y:8s                 ' –128 to 127
DIM flag:1               ' single-bit variable (0 or 1)
DIM x                    ' properties defined by Compiler Options
DIM x:s                  ' signed variable, size defined by
                         ' Compiler Options
```

EEPROM <address>,<constant>|<string>{,<constant>|<string>,...}

See **DATA**.

END

Terminates a BASIC program. Tristates all pins (high impedance state) and stops execution.
This instruction is optional in a program. If execution reaches the last statement it will stop anyway.

```
IF !button THEN END       ' terminate program if button pressed
```

FOR <variable> = <start> **TO** <end> {**STEP** <step>}

Repeats a sequence of instructions. Instructions between the **FOR** statement and the matching **NEXT** statement are executed one or more times.

<variable> is initialised to the value <start>, instructions in the loop are executed, <step> is added to <variable>, which is then compared to <end>. The loop is repeated until <variable> is greater than <end> if <step> is positive, or until <variable> is less than <end> if <step> is negative. The step is optional and defaults to 1 but can be any positive or negative constant or constant expression. <start> and <end> can be arbitrary expressions. <start> is evaluated once at the beginning of the loop, and <end> is evaluated each time through the loop. Note that the loop always executes at least once irrespective of the values of <start> and <end>.

Each **FOR** statement should be matched by a single **NEXT** statement.

See also **NEXT**.

```
        FOR n = 1 TO 0                ' this loop executes once
        ....
        NEXT

        FOR n = 10 TO 1 STEP -1       ' this loop executes 10 times
        ....
        NEXT

        FOR n = 1 TO 10               ' this loop executes 5 times
        ....
        n = n + 1
        NEXT n

        FOR x = 1 TO 10               ' 10 times through outer loop
        FOR y = 1 TO 10 STEP 2        ' 10 * 5 times through inner loop
        ....
        NEXT y
        NEXT x

        LET <variable> = <start>      ' FOR loop equivalent
loop:   ....
        LET <variable> = <variable> + <step>
        IF <variable> <= <end> THEN loop
```

GOSUB <label>

Executes a subroutine. When the subroutine ends (with a **RETURN** instruction) execution continues with the instruction following the **GOSUB**. Subroutines are useful for sharing common code and for logically dividing a program up into chunks.

Subroutines can be nested (a subroutine calling another subroutine) but they should not be nested more than about 16 levels. If this limit is exceeded then the program is halted and the stack overflow is indicated by the onboard LED flashing. Clear the error condition by unplugging then reconnecting the USB cable.

See also **RETURN**.

```
        GOSUB flash             ' simple subroutine call
        ....
flash:  HIGH led
        PAUSE 100
        LOW led
        RETURN

        GOSUB sub1              ' nested subroutines
        ....
sub1:   GOSUB sub2              ' max depth < 20
        ....
        RETURN
sub2:   ....
        RETURN
```

GOTO <label>

Unconditional branch. Execution continues with the instruction at <label>.

```
           GOTO timeout
           ....
timeout:   PRINT "*** Timeout! ***"
           END
```

HIGH <pin>|<variable>

Makes a pin high, or sets a variable to 1.

<pin> can either be a pin name, or a number from 1 to 11 (not 12 as pin12 is input only). Alternatively a variable can be specified in which case this instruction is equivalent to **LET** <variable> = 1. If a variable is used in this way before it is defined (see **DIM**) then it is created as a single-bit variable.

SET is a synonym for this instruction. See also **LOW**, **TOGGLE**.

```
HIGH led                        ' turn onboard LED on
HIGH 3                          ' make pin3 high

HIGH x                          ' x = 1

LET n = 5
HIGH n                          ' this is not the same as HIGH 5
```

IF <expression> **THEN** <label>|{**THEN**} <instruction> {**ELSE** <label>|<instruction>}

Conditional branching or execution. <expression> is evaluated and depending on whether it is true or false (non-zero or zero) the program branches to <label> or executes <instruction>. The **ELSE** part optionally specifies an alternative branch or instruction to be executed if the **THEN** part is false.

Note that if the conditional instruction is an assignment then the keyword **LET** is required.

Take care with operator precedence in expressions. For example, **IF** mask & 3 = 0 **THEN** ... The = operator has a higher precedence than the & operator. To make the instruction execute as expected bracket it thus **IF** (mask & 3) = 0 **THEN** ...

```
IF n > 10 LET n = 0             ' wrap around if > 10

IF n > 100 PRINT "too big!"

IF n THEN label                 ' branches if n <> 0

IF a & b THEN label             ' might not work (1 & 2 = 0 e.g.)
IF a <> 0 & b <> 0 THEN label   ' correct form

IF x > 123 THEN exit            ' these three statements are
IF x > 123 THEN GOTO exit       ' identical
IF x > 123 GOTO exit

IF flag THEN LET x = x + 1      ' LET is required
IF flag LET x = x + 1           ' shorter format
IF flag x = x + 1               ' causes a syntax error

IF x < 0 THEN not_ok ELSE ok

IF x >= 0 PRINT "positive" ELSE PRINT "negative"

IF mask & 3 = 0 THEN ...        ' actually IF mask & (3 = 0)
                                ' or IF mask & 0
```

INPUT <pin>{,**off**|**pullup**}

Configures a pin as a digital input.

<pin> can either be a pin name, or a number from 1 to 11, or also 12 if Master Clear is disabled in PIC Options.

The second argument controls an internal weak pull-up resistor and defaults to off. These pull-up resistors make connecting to pushbuttons and switches easier. Simply connect the button or switch between the pin and ground. The pull-up resistor holds the input high until the button is pressed which pulls it to ground. You should therefore check the pin for a low to indicate the button is pressed.

The state of a digital pin is tested using the pin names *pin1*, *pin2* etc.

Note that pull-up resistors are only available on pins 3, 4, 5, 7 and 12. Also note that pins 1, 2, 6, 8, 9, 10 and 11 are implemented with Schmitt trigger inputs which affects the voltage level at which they switch (see the PIC documentation at www.microchip.com for further details).

```
INPUT 1                     ' pin1 digital input
INPUT pin3                  ' pin3 digital input, no pull-up
INPUT pin4,pullup           ' pin4 digital input, weak pull-up
IF pin5 THEN ...            ' if pin5 is high ...
```

LCD <pin_d7>,<pin_d6>,<pin_d5>,<pin_d4>,<pin_rs>,<pin_rw>,<pin_e>,<rows>,<cols>

Configures an LCD device. Hitachi HD44780 and compatible modules are supported. This is the most common type of display and is a de facto standard.

Seven pins are required to control an LCD device. The first four arguments specify the pins to connect to the LCD data bus. The LCD is operated in 4-bit mode so only four data lines are needed (high nibble). The next three pins specify the register select, read/write and enable lines.

The pins can either be pin names or numbers from 1 to 11.

The final two arguments specify the number of rows and columns of the display.

See also **SEND LCD**, **PRINT LCD**, **CLS LCD**.

```
LCD pin1,pin2,pin3,pin4,pin5,pin6,pin7,2,16    ' initialise LCD

PRINT LCD "message"                            ' display message
PRINT LCD x                                    ' display variable

CLS LCD                                        ' clear display

LCD                                            ' LCD off
```

{**LET**} <variable> = <expression>

Sets a variable to a value. The keyword is optional except in **IF** statements where it is required. <expression> can be an arbitrary expression involving constants, operators and brackets. If a variable is used before being defined (see **DIM**) then its size and signed properties default to those specified in the Compiler Options.

```
LET x = 0                   ' clear variable
x = 0                       ' LET is optional

x = x + 1                   ' increment variable
IF flag LET x = x + 1       ' LET is required

LET n1 = $ab                ' hexadecimal constant
LET n2 = %10101010          ' binary constant
LET n3 = "*"                ' character constant
LET n4 = "\""               ' embedded double quote
LET n5 = "\\"               ' embedded backslash

LET z = 10*x + y/2          ' complex expression
```

LOW <pin>|<variable>

Makes a pin low, or zeroes a variable.

<pin> can either be a pin name, or a number from 1 to 11 (not 12 as pin12 is input only). Alternatively a variable can be specified in which case this instruction is equivalent to **LET** <variable> = 0. If a variable is used in this way before it is defined (see **DIM**) then it is created as a single-bit variable.

RESET and **CLEAR** are synonyms for this instruction. See also **HIGH**, **TOGGLE**.

```
LOW led                    ' turn onboard LED off
LOW pin4                   ' make pin4 low

LOW x                      ' x = 0

SYMBOL n = 5
LOW n                      ' this is the same as LOW 5
```

MOTORS <pin1A>,<pin1B>,<pin2A>,<pin2B>{,**off**|**on**}

Configures a pair of DC motors.

The first four arguments specify the output pins to use. These pins are typically connected to the input pins of an H-bridge driver chip. Pins from 1 to 11 are available.

The first motor is connected to the 1A and 1B outputs of the H-bridge, and the second motor to the 2A and 2B outputs.

The fifth argument is optional and defaults to **off**.

The motors speed can be set using the variables *motor1* and *motor2*. The speeds are specified between -100 and 100, with the former representing full speed in reverse, and the latter full speed forward.

It is generally not a good idea to instantaneously switch the motor speed from off to full. A better approach is to ramp up the speed in steps slowly to the target speed. This reduces the chance of a power surge crashing the processor.

Note that when motors are first configured the speeds are zero.

```
MOTORS pin1,pin2,pin3,pin4,on      ' motors on pin1,pin2,pin3,pin4
MOTORS                             ' motors off
LET motor1 = 0                     ' motor 1 off
motor2 = 100                       ' motor 2 full speed forward
motor1 = -50                       ' half speed reverse
IF motor2 = 0 THEN ...             ' if motor stopped ...
```

NEXT {<variable>}

Marks the end of a **FOR** loop. The variable can be omitted only for non-nested loops, in which case **NEXT** matches the last **FOR** statement.

Each **FOR** statement should be matched by a single **NEXT** statement.

See also **FOR**.

```
        FOR n = 1 TO 10          ' 10 times through the loop
        PRINT n
        NEXT n                   ' n could be omitted
        FOR x = 1 TO 10          ' nested loops
        FOR y = 1 TO 10
        ....
        IF error THEN cont       ' exit inner loop prematurely
        ....
        NEXT y
cont:   NEXT x
```

ON <expression> **GOTO**|**GOSUB** <label>{,<label>,...}

Branches to one of a number of labels, or calls one of a number of subroutines. If the expression evaluates to greater than the number of labels then execution continues at the next instruction. The number of labels is limited to 32, and the index starts at 0.

See also **GOTO**, **GOSUB**.

```
        ON state GOSUB lbl1,lbl2,lbl3
        ....
lbl1:   HIGH pin1               ' if state = 0
        RETURN
lbl2:   HIGH pin2               ' if state = 1
        RETURN
lbl3:   HIGH pin3               ' if state = 2
        RETURN
```

OUTPUT <pin>

Configures a pin as a digital output.

<pin> can either be a pin name, or a number from 1 to 11 (not 12 as pin12 is input only).

See also **HIGH**, **LOW**, **TOGGLE**.

```
OUTPUT pin1                    ' pin1 digital output
HIGH pin1                      ' pin high
OUTPUT 2                       ' pin2 digital output
LOW 2                          ' pin low
```

PAUSE <duration>

Pauses for a number of milliseconds (thousandths of a second). The duration can be a constant or an expression. The accuracy of the pause is 1ms - **PAUSE** 1 might actually delay for less than 1ms. The maximum pause is 65535ms (a little over a minute).

Timers and interrupts are checked during a pause but pauses cannot be nested. In other words if a timer or interrupt subroutine is called while a **PAUSE** instruction is executing, then the subroutine cannot itself use the **PAUSE** instruction.

If the duration is omitted then the pause is infinite.

```
        PAUSE 500                               ' wait half a second
        PAUSE                                   ' wait for ever
        DIM start:16u                           ' PAUSE equivalent code
        start = ticks                           ' if timer or interrupt
wait:   IF ticks – start < 1000 GOTO wait       ' subroutine uses PAUSE
```

PRINT LCD|**SPI**|**UART**|**USB** <expression>|<string>{,|;<expression>|<string>,|;...}

Prints characters on an output device.

PRINT is similar to **SEND** but formats its output whereas **SEND** transmits raw data.

PRINT USB works in conjunction with the Console window and is a useful aid to debugging as it allows variables to be examined etc. Note that **PRINT** is a shortcut for **PRINT USB**.

The following print modifiers are available: %(x) to print binary, $(x) to print hex, and @(x) or #(x) to print a character. The default is to print in decimal.

PRINT can print multiple arguments with a single statement. If the arguments are separated with commas (,) then tabs are inserted between them. If they are separated with semicolons (;) then arguments are printed next to each other with no intervening space.

PRINT outputs a newline after printing its arguments (unless the statement ends with a comma or semicolon). You can use **PRINT** "\n" to force additional newlines.

Note that numbers greater than 32767 will print as negative numbers, and numbers less than -32768 will print as positive numbers.

```
PRINT LCD x;" squared is ";x*x

PRINT USB $(hex)              ' display variable in hexadecimal

PRINT LCD %(255)              ' prints "%11111111"
PRINT LCD %(11)               ' prints "%00001011" (decimal 11
                              ' in binary)
PRINT UART %(%11)             ' prints "%00000011"
PRINT UART $(255)             ' prints "$FF"
PRINT UART #(65)              ' prints "A"

PRINT "no";"space"            ' prints concatenated strings

PRINT USB "col1\tcol2"        ' prints in two columns (\t = tab)
PRINT USB "col1","col2"       ' also prints in two columns
PRINT USB "line1\nline2"      ' prints on two lines (\n = newline)

PRINT "some text\r\n";        ' outputting to a device that
                              ' requires two characters at the
                              ' end of each line
```

PWM <pin>**,**<period>**,**<duty>

Configures a pulse width modulation output signal. A PWM signal is a square wave with variable period (frequency) and variable 'duty cycle'. The duty cycle is the fraction of each cycle that the output is high. A duty cycle of 0% means that the output is never high. A duty cycle of 100% means that the output is always high. A duty cycle of 50% means that the output is high half the time and low half the time (i.e. a symmetrical square wave).

The first argument specifies the output pin to use. Pins 1 to 11 are available.

The second argument specifies the PWM period. The period is expressed in ms or milliseconds (thousandths of a second) and ranges from 1 to 65535. The PWM period is obtained from the relationship <period> = 1000/<frequency>.

A period of 0 or an unspecified period disables the PWM output.

The third argument specifies the initial duty cycle. The duty cycle is expressed as a percentage between 0 and 100.

The duty cycle can also be subsequently controlled using the variables *pwm1*, *pwm2* etc.

PWM signals are useful for simulating variable level analog output voltages. A typical use is controlling the brightness of an LED. A period of 10ms or less is suggested when driving an LED to avoid visible flicker.

A long PWM period could be used to flash an LED (at full brightness). For example a period of 2000 would cause an LED to flash once every 2 seconds.

PWM signals can also be used for controlling the speed of a DC motor (driven in one direction only). Additional hardware would be required as motors draw more current than a PIC output pin can deliver. The simplest method is a transistor or Darlington output driver.

```
PWM pin1,10,50     ' PWM on pin1, period = 10ms, duty cycle = 50%
PWM 2,50,5         ' PWM on pin2, frequency = 20Hz, duty cycle = 5%

PWM pin3,0         ' PWM on pin3 off
PWM 4              ' PWM on pin4 off

LET pwm1 = 0       ' PWM output on pin1 continuously low
pwm2 = 100         ' PWM output on pin2 continuously high
pwm3 = 50          ' 50% duty cycle output
```

RANDOM {<variable>}

Gets the next number in a pseudo random sequence. The random number is also stored in the 16-bit variable *rand*.

You can seed the random number sequence by setting *rand* to a particular value.

```
DIM r:16u
RANDOM r
PRINT "The next random number is ";r
PRINT "The last random number was ";rand
PRINT "Random number between 0 and N inclusive = ";rand % (N+1)
```

READ <address>,<variable>{,<variable>,...}

Reads 8-bit data from EEPROM data memory. EEPROM memory is non-volatile which means that its contents are retained after power is removed. It can be used to store data that is needed permanently, or as additional memory for variables etc.

<address> is an expression that evaluates to an address in the range 0 to 239. An address outside this range reads as zero. Data is read from consecutive memory locations into the specified variable(s).

See also **WRITE**, **DATA**.

```
DIM x:16u,high:8u,low:8u    ' read a 16-bit variable as two bytes
READ 10,high,low
LET x = (high<<8)|low       ' brackets needed because of precedence
```

RECEIVE SPI <variable>

Receives 8-bit data from the SPI serial port. The received data is stored in the specified variable.

RECEIVE SPI can only receive a single byte at a time without an intervening **SEND SPI**.

If no data is available then this instruction does not wait but returns null. Because the SPI module acts as an SPI master, new data will only become available after the next **SEND SPI**. The variable *spi_in* can be used to check the port status before receiving. This flag variable is set if a byte is available or clear if not.

The variable *error* can be used to check for errors after receiving. If an error has occurred then *error* will be non-zero.

See also **SEND SPI**.

```
IF !spi_in GOTO no_data     ' byte received ?
RECEIVE SPI x               ' read it if yes
IF error GOTO rx_error      ' branch if receive error
```

RECEIVE UART <variable>{,<variable>,...}

Receives 8-bit data from the UART serial port (RS232). The received data is stored in the specified variable(s). For word variables only the lower byte is affected.

Note that **RECEIVE UART** does not timeout but waits (perhaps indefinitely) until a byte has been received. The variable *uart_in* can be used to check the port status before receiving. This flag variable is set if a byte is available or clear if not.

The variable *error* can be used to check for errors after receiving. If an error has occurred then *error* will be non-zero.

See also **SEND UART**.

```
IF !uart_in GOTO no_data    ' byte received ?
RECEIVE UART x              ' read it if yes
IF error GOTO rx_error      ' branch if receive error

DIM res:16u,high:8u,low:8u  ' receiving word data -
RECEIVE UART high,low       ' assemble from two bytes
LET res = (high<<8)|low     ' brackets needed because of precedence
```

RECEIVE USB <variable>{,<variable>,...}

Receives 8-bit data from the USB port. The received data is stored in the specified variable(s). For word variables only the lower byte is affected.

Note that **RECEIVE USB** does not timeout but waits (perhaps indefinitely) until a byte has been received. The variable *usb_in* can be used to check the port status before receiving. This flag variable is set if a byte is available or clear if not.

The variable *error* can be used to check for errors after receiving. If an error has occurred then *error* will be non-zero.

If the Console window is open then keys pressed on the keyboard will be sent to the USB port. But equally data can be sent by any other application that has opened the virtual COM port that maps to the USB connection.

See also **SEND USB**.

```
IF !usb_in GOTO no_data     ' byte received ?
RECEIVE USB x               ' read it if yes
IF error GOTO rx_error      ' branch if receive error

DIM res:16u,high:8u,low:8u  ' receiving word data -
RECEIVE USB high,low        ' assemble from two bytes
LET res = (high<<8)|low     ' brackets needed because of precedence
```

REM

Indicates a comment (remark). Comments are aids to the understanding of a program and are ignored by the compiler. **REM** and **'** can be used on lines by themselves, and **'** can be used after other instructions.

```
REM this is a comment
        IF x > 10 GOTO too_big      ' test x against upper limit
        LET x = 1                   REM this will cause a syntax error
```

RESET <pin>|<variable>

See **LOW**.

RETURN

Returns from a subroutine. Execution continues with the instruction following the corresponding **GOSUB**.

See also **GOSUB**.

```
        GOSUB flash
        ....                                ' execution continues here
flash:  TOGGLE led
        RETURN
```

SEND LCD <expression>|<string>{,<expression>|<string>,…}

Sends data to the LCD display. The data to be transmitted can be expressions or strings.
Only the low 8 bits of each expression is sent, and no zero-terminator is sent at the end of strings.
This instruction could be useful for displaying non-ASCII characters.
See also **PRINT LCD** which is more efficient at sending strings and also offers formatting options.

```
SEND LCD 65,66,67           ' "ABC"
SEND LCD "hello world\n"
```

SEND SPI <expression>|<string>{,<expression>|<string>,…}

Sends data to the SPI serial port. The data to be transmitted can be expressions or strings.

Only the low 8 bits of each expression is sent, and no zero-terminator is sent at the end of strings.

Note that this instruction waits until the data has been sent before completing.

Because the SPI module acts as an SPI master, the act of sending a byte receives a byte at the same time.

PRINT SPI is more efficient at sending strings and also offers formatting options.

See also **RECEIVE SPI**.

```
SEND SPI 12,34,56,flag      ' transmits 8-bit bytes
SEND SPI "hello world\n"    ' transmits the string followed
                            ' by a newline
DIM x:16                    ' 16-bit word
SEND SPI x                  ' only transmits low byte
SEND SPI x>>8,x&$ff         ' transmits word (high byte then low)
```

SEND UART <expression>|<string>{,<expression>|<string>,…}

Sends data to the UART serial port (RS232). The data to be transmitted can be expressions or strings.

Only the low 8 bits of each expression is sent, and no zero-terminator is sent at the end of strings.

Note that this instruction waits until the data has been accepted before completing. The variable *uart_out* however can be checked prior to sending data. This flag variable will be set if the UART is ready to receive the next byte.

PRINT UART is more efficient at sending strings and also offers formatting options.

See also **RECEIVE UART**.

```
SEND UART 12,34,56,flag     ' transmits 8-bit bytes
IF uart_out SEND UART 123   ' send if UART ready
SEND UART "hello world\n"   ' transmits the string followed
                            ' by a newline
DIM x:16                    ' 16-bit word
SEND UART x                 ' only transmits low byte
SEND UART x>>8,x&$ff        ' transmits word (high byte then low)
```

SEND USB <expression>|<string>{,<expression>|<string>,…}

Sends data to the USB port. The data to be transmitted can be expressions or strings.

Only the low 8 bits of each expression is sent, and no zero-terminator is sent at the end of strings.

Note that this instruction waits until the data has been accepted before completing. The variable *usb_out* however can be checked prior to sending data. This flag variable will be set if the USB is ready to receive the next byte.

If the Console window is open then data will appear in this window. But equally data can be received by any other application that is listening on the virtual COM port that maps to the USB connection.

PRINT USB is more efficient at sending strings and also offers formatting options.

See also **RECEIVE USB**.

```
SEND USB 12,34,56,flag      ' transmits 8-bit bytes
IF usb_out SEND USB 123     ' send if USB ready
SEND USB "hello world\n"    ' transmits the string followed
                            ' by a newline
DIM x:16                    ' 16-bit word
SEND USB x                  ' only transmits low byte
SEND USB x>>8,x&$ff         ' transmits word (high byte then low)
```

SERVO <pin>{,**off**|**on**}

Configures a servo motor.

The first argument specifies the output pin to use. Pins 1 to 11 are available.

The second argument is optional and defaults to **off**.

Servo motors are controlled by means of a pulse-width modulated signal. The width of the pulse (the duty cycle) determines the position of the servo. For a standard servo, pulses of 1.5ms centre it, pulses of 1.0ms move it 90° anticlockwise, and pulses of 2.0ms move it 90° clockwise. Particular servos may have different maximum & minimum angles and control pulse ranges. The absolute limit to the range is -180° (0.5ms) to +180° (2.5ms).

Servos have three wires which are normally coloured as follows: red = positive power, black = negative ground, blue or white = control. The power lines should be connected to the power connections on the *JellyBean* board or to an external power supply (with its ground commoned with the *JellyBean* ground). The control wire should be connected to a pin (1 to 11).

The servo position (angle) can be set using the variables *servo1*, *servo2* etc. A value of 0 represents the servo centre position; a value of 100 represents 180° clockwise; and a value of -100 represents 180° anticlockwise.

The current position can also be read using the same variable, but bear in mind that the variable represents the target servo position rather than its instantaneous position. There is a practical limit to how fast a servo can actually move to a new position. Changing the angle from say 0 to 90 degrees doesn't result in an instantaneous change of position. The tracking speed depends on the particular servo in use.

Note that when a servo is configured the angle is set to zero.

```
SERVO pin1,on             ' servo on pin1
SERVO 2,on                ' servo on pin2

SERVO pin3,off            ' servo on pin3 off
SERVO 4,0                 ' servo off
SERVO pin5                ' servo off

LET servo1 = 0            ' centre servo on pin1
servo2 = -50              ' 90° anticlockwise
servo3 = 25               ' 45° clockwise
IF servo4 = 0 THEN ...    ' if servo centred ...
```

SET <pin>|<variable>

See **HIGH**.

SPI <clock>,<mode>

Configures the SPI serial port.

The first argument specifies the clock rate. Three speeds are available: 0 = master clock/64 (750kHz); 1 = master clock/16 (3MHz); 2 = master clock/4 (12MHz).

The second argument specifies the bus mode (0 to 3). The four standard SPI modes are available: 0 = clock idle low, read data on rising edge; 1 = clock idle low, read data on falling edge; 2 = clock idle high, read data on falling edge; 3 = clock idle high, read data on rising edge.

If no arguments are specified the SPI module is disabled.

The SPI uses the fixed pins 6 to transmit, 3 to receive, and 4 for the clock. The module operates as an SPI master in 8-bit mode.

See also **RECEIVE SPI**, **SEND SPI**.

```
SPI 1,2                         ' configure SPI

SEND SPI x                      ' send byte
RECEIVE SPI y                   ' receive byte

SPI                             ' SPI off
```

STEPPER <pin1A>,<pin1B>,<pin2A>,<pin2B>,<speed>{,**off**|**standard**|**high_torque**|**half_step**}

Configures a stepper motor.

The first four arguments specify the output pins to use. These pins are typically connected to the input pins of an H-bridge driver chip. Pins from 1 to 11 are available.

The output pins of the H-bridge connect to the (bipolar or unipolar) stepper motor itself. The first stepper coil of a bipolar stepper is connected to 1A & 1B, and the second coil to 2A & 2B. A unipolar stepper is driven by grounding or not connecting the centre taps.

The fifth argument sets the stepper tracking speed in terms of milliseconds per step cycle. The fastest tracking speed is a minimum delay of 1ms between steps, and the slowest 255ms between steps. Note that there is a physical limit to the tracking speed for every stepper motor. If the maximum for a particular stepper is exceeded then it will cease to rotate properly.

The sixth argument specifies a stepper driving sequence and defaults to **off**. The sequences are:

Standard sequence: `1A -> 2A -> 1B -> 2B`
High-torque sequence: `1A+2A -> 2A+1B -> 1B+2B -> 2B+1A`
Half-step sequence: `1A+2A -> 2A -> 2A+1B -> 1B -> 1B+2B -> 2B -> 2B+1A -> 1A`

where `1` is the first coil and `2` the second, and `A` and `B` are the two connections to each coil (i.e. `1A` connected to the first H-bridge output pin, `1B` to the second, `2A` to the third, and `2B` to the fourth H-bridge pin). Note that the high-torque sequence consumes twice the current because both coils are energised at the same time, and the tracking speed of the half-step sequence is half that of the other two sequences.

The standard sequence should be suitable for most situations.

The stepper position can be set using the variable *stepper1*. The position is specified between -32768 and 32767, with the former representing the most extreme anticlockwise position and the latter the most extreme clockwise position.

The position is measured in steps rather than in degrees. The exact relationship between steps and angle depends on the particular stepper in use, but a step is typically a few degrees.

Continuous rotation is possible, but eventually the limit of -32768 or +32767 will be reached. In order to accommodate this situation the stepper can be periodically 'zeroed' by reconfiguring it (i.e. repeating the initial configuration statement).

The current position can also be read using the same variable, but bear in mind that the variable represents the target stepper position rather than its instantaneous position. There is a practical limit to how fast a stepper can actually move to a new position. The tracking speed depends on the particular stepper in use.

Note that each time a stepper is configured or re-configured the position is set to zero.

```
STEPPER 1,2,3,4,100,standard  ' standard stepper on pin1,pin2,
                              ' pin3,pin4 with speed 100
STEPPER                       ' stepper off
LET stepper1 = 0              ' move stepper to initial position
stepper1 = 20                 ' 20 steps clockwise from start
stepper1 = -50                ' 50 steps anticlockwise from start
stepper1 = stepper1 + 1       ' move one step clockwise (relative
                              ' position)
stepper1 = stepper1 - 10      ' move 10 steps anticlockwise
IF stepper1 = 0 THEN ...      ' if stepper at initial position ...
```

SYMBOL <symbol> = <constant>|<variable>

Defines a constant or an alias for a variable or pin. **SYMBOL** definitions should be placed prior to any references to them. The use of symbols can make a program easier to understand and reduce the chance of errors.

```
SYMBOL LIMIT = 100            ' upper limit
IF x > LIMIT GOTO too_big

LET x = N                     ' backward reference will cause
SYMBOL N = 10                 ' an error

SYMBOL SIZE = 10*20           ' constant expression

SYMBOL switch = pin2          ' define an alias
IF !switch THEN pressed

SYMBOL alert = led            ' define an alias
HIGH alert
```

TOGGLE <pin>|<variable>

Makes a pin high if it is currently low or low if it is currently high, or toggles a variable between 0 and 1.

<pin> can either be a pin name, or a number from 1 to 11 (not 12 as pin12 is input only). Alternatively a variable can be specified in which case this instruction is equivalent to **LET** <variable> = <variable> ^ 1 when <variable> is a single bit. If a variable is used in this way before it is defined (see **DIM**) then it is created as a single-bit variable.

See also **HIGH**, **LOW**.

```
TOGGLE led              ' turn LED off if on and vice versa
TOGGLE 5                ' toggle pin5
TOGGLE x                ' x = 0 if non-zero or x = 1 if zero
DIM flag:1              ' equivalent to flag = flag ^ 1 but
TOGGLE flag             ' more efficient
```

UART <baud>,<data>,<parity>,<stop>

Configures the UART serial port (RS232) for asynchronous communications.

The four arguments specify the baud rate (bits per second), the number of data bits (7 or 8), the parity bit (none, even or odd), and the number of stop bits (1 or 2). The Wizard default settings are 9600 baud, 8 data bits, no parity bit, and 2 stop bits. Generally only the baud rate would need to be changed from the defaults as the other settings are very standard. Note that the combination 8 data bits + parity + 2 stop bits is not available.

The baud rate must be specified as a literal constant and not as a symbol or variable.

If no arguments are specified the UART is disabled.

The UART uses the fixed pins 7 to transmit and 5 to receive. RS232 handshaking is not directly supported by the hardware, but can easily be implemented using a pair of I/O pins.

If connecting to the RS232 serial port of a PC then it is usually necessary to add a level converter chip such as a MAX232. This is because RS232 communications work at 12V levels whereas the PIC works at 5V levels. Some PC hosts tolerate 5V levels and might work reliably but not using a converter chip could damage the PIC.

If connecting to other hardware operating at 5V levels then a converter chip is not required.

The UART transmit and receive format is standard non-return-to-zero (NRZ) with idle = high, start bit = low, '0' data bit = low, '1' data bit = high, stop bit(s) = high.

See also **RECEIVE UART**, **SEND UART**.

```
UART 19200,7,even,1           ' configure UART – 19200 bits per
                              ' second, 7 data bits + even parity
                              ' bit, 1 stop bit
UART 31250,8,none,1           ' MIDI output
IF !uart_in GOTO no_data      ' byte received ?
RECEIVE UART x                ' read it if yes
SEND UART y,123               ' sending bytes
UART                          ' UART off
```

WHEN <pin> **= 0|1 GOSUB** <label>

Interrupt processing. The subroutine at <label> is called when the input pin is low (0) or high (1).

If the interrupt condition remains true then the subroutine will be called repeatedly. The subroutine for an interrupt can be interrupted by an interrupt on a different pin.

Interrupt processing is synchronous with the main program. This means that the interrupt subroutine can only be executed between individual BASIC statements in the main program rather than at some arbitrary time.

Interrupts can be enabled on pins 1 to 11, and also pin 12 if Master Clear is disabled in PIC Options. Only a single interrupt is available per pin. **WHEN** <pin> disables interrupt processing for a pin.

To enable interrupt processing the 'Timers and Interrupts' setting in Compiler Options must be checked. There is a performance penalty associated with this setting so it shouldn't be enabled if timers and interrupts are not being used.

An interrupt subroutine should normally be exited with a **RETURN** statement. You can return to the main program with a **GOTO** statement but you should reconfigure the interrupt if doing so. To prevent stack overflows you should also reset the stack pointer as in the example below, and also clear any pause if the interrupt event occurred during a **PAUSE** statement.

See also **GOSUB**.

```
        WHEN pin1 = 1 GOSUB mysub
        ....
mysub:  ....                        ' this subroutine is called
        RETURN                      ' when pin1 is high

        WHEN pin2 = 0 GOSUB sub1    ' interrupt if pin2 is low
        ....
sub1:   HIGH LED
        WHEN pin2                   ' disable interrupt
        RETURN

        WHEN pin2 = 1 GOSUB sub2    ' configure interrupt initially
loop:   ....                        ' main program loop
        GOTO loop
sub2:   ....
        CLEAR stack                 ' reset stack pointer
        PAUSE 0                     ' clear pause
        WHEN pin2 = 1 GOSUB sub2    ' reconfigure interrupt
        GOTO loop                   ' continue with main program
```

WHEN <timer> = <period> **GOSUB** <label>

Background processing. The subroutine at <label> is called repeatedly every <period>/1000 seconds.

Background processing is synchronous with the main program. This means that the background subroutine can only be executed between individual BASIC statements in the main program rather than at some arbitrary time.

<timer> can be one of four timers, and <period> is specified in thousandths of a second (milliseconds) in the range 1 to 65535. **WHEN** <timer> disables background processing for a timer.

If the background subroutine uses variables altered by the main program then use a semaphore to guarantee exclusive access.

To enable background processing the 'Timers and Interrupts' setting in Compiler Options must be checked. There is a performance penalty associated with this setting so it shouldn't be enabled if timers and interrupts are not being used.

A background subroutine should normally be exited with a **RETURN** statement. You can return to the main program with a **GOTO** statement but you should reconfigure the timer if doing so. To prevent stack overflows you should also reset the stack pointer as in the example below, and also clear any pause if the timer event occurred during a **PAUSE** statement.

See also **GOSUB**.

```
        WHEN timer1 = 500 GOSUB flash
        ....
flash:  TOGGLE led                  ' this subroutine is called
        RETURN                      ' every half second

        WHEN timer2                 ' timer 2 off

        DIM semaphore:1             ' define semaphore
        WHEN timer1 = 100 GOSUB sub1
        SET semaphore
        ....                        ' critical code
        CLEAR semaphore
        ....
sub1:   IF semaphore RETURN         ' this subroutine is called
        ....                        ' 10 times every second
        RETURN

        WHEN timer2 = 100 GOSUB sub2  ' configure timer initially
loop:   ....                        ' main program loop
        GOTO loop
sub2:   ....
        CLEAR stack                 ' reset stack pointer
        PAUSE 0                     ' clear pause
        WHEN timer2 = 100 GOSUB sub2  ' reconfigure timer
        GOTO loop                   ' continue with main program
```

WRITE <address>,<expression>|<string>{,<expression>|<string>,...}

Writes 8-bit data to EEPROM data memory. EEPROM memory is non-volatile which means that its contents are retained after power is removed. It can be used to store data that is needed permanently, or as additional memory for variables etc.

<address> is an expression that evaluates to an address in the range 0 to 239. An address outside this range is ignored. The data to be written can be an expression or a string. Data is stored in consecutive memory locations, and strings are stored without a zero terminator.

See also **READ**, **DATA** (this instruction differs from **DATA** in that the data is written each time the instruction is executed).

```
WRITE 0,high_score

WRITE 1,"a character string"   ' no string terminator
WRITE 1,"another string",0     ' zero terminator

DIM x:16
WRITE 10,x                     ' only stores low byte
WRITE 10,x>>8,x&$ff            ' stores a complete 16-bit variable
```

Operators and Precedence

BASIC expressions consist of constants, variables and the following operators. Operators with higher precedence are evaluated before those with lower precedence. Unary operators of equal precedence are evaluated right to left, and binary operators of equal precedence are evaluated left to right. Brackets can be used to override the precedence order.

()	brackets	10	highest precedence
+	unary plus	9	no operation
-	unary minus	9	negation (two's complement)
!	logical NOT (unary)	9	!x = 1 if x = 0 otherwise !x = 0
~	bitwise complement (unary)	9	~0 = 1, ~1 = 0 (one's complement)
*****	multiply	8	signed or unsigned multiplication
/	divide	8	signed or unsigned integer division
%	modulus	8	signed or unsigned remainder
+	add	7	signed or unsigned addition
-	subtract	7	signed or unsigned subtraction
>>	shift right	6	>>n equivalent to dividing by 2 to the power of n
<<	shift left	6	<<n equivalent to multiplying by 2 to the power of n
>=	greater than or equal	5	1 if true, 0 if false
>	greater than	5	1 if true, 0 if false
<=	less than or equal	5	1 if true, 0 if false
<	less than	5	1 if true, 0 if false
=	equal	4	1 if true, 0 if false
<>	not equal	4	1 if true, 0 if false
&	bitwise AND	3	0 & 0 = 0, 0 & 1 = 0, 1 & 0 = 0, 1 & 1 = 1
^	bitwise XOR	2	0 ^ 0 = 0, 0 ^ 1 = 1, 1 ^ 0 = 1, 1 ^ 1 = 0
|	bitwise OR	1	0 | 0 = 0, 0 | 1 = 1, 1 | 0 = 1, 1 | 1 = 1

Shift right and shift left are limited to shifts from 0 to 16 positions, and the shift must be a constant. Shift right is an arithmetic shift for signed variables, or a logical shift for unsigned variables.

Appendix A - System Variables

Variables predefined by the compiler and available to user programs.

Variable	Bits	Initial	Range	Notes
pin1	1	-	0 or 1	pin 1
pin2	1	-	0 or 1	pin 2
pin3	1	-	0 or 1	pin 3
pin4	1	-	0 or 1	pin 4
pin5	1	-	0 or 1	pin 5
pin6	1	-	0 or 1	pin 6
pin7	1	-	0 or 1	pin 7
pin8	1	-	0 or 1	pin 8
pin9	1	-	0 or 1	pin 9
pin10	1	-	0 or 1	pin 10
pin11	1	-	0 or 1	pin 11
pin12	1	-	0 or 1	pin 12
led [(1)]	1	0	0 or 1	onboard LED, 1 = on
button [(2)]	1	-	0 or 1	onboard pushbutton, 0 if pressed
analog1	16	-	0 to 1023	analog input on pin 1
analog2	16	-	0 to 1023	analog input on pin 2
analog3	16	-	0 to 1023	analog input on pin 3
analog5	16	-	0 to 1023	analog input on pin 5
analog6	16	-	0 to 1023	analog input on pin 6
analog8	16	-	0 to 1023	analog input on pin 8
analog9	16	-	0 to 1023	analog input on pin 9
pwm1	8	0	0 to 100	PWM 1 duty cycle
pwm2	8	0	0 to 100	PWM 2 duty cycle
pwm3	8	0	0 to 100	PWM 3 duty cycle
pwm4	8	0	0 to 100	PWM 4 duty cycle
pwm5	8	0	0 to 100	PWM 5 duty cycle
pwm6	8	0	0 to 100	PWM 6 duty cycle
pwm7	8	0	0 to 100	PWM 7 duty cycle
pwm8	8	0	0 to 100	PWM 8 duty cycle
pwm9	8	0	0 to 100	PWM 9 duty cycle
pwm10	8	0	0 to 100	PWM 10 duty cycle
pwm11	8	0	0 to 100	PWM 11 duty cycle
timer1	16	0	0 to 65535	background processing timer 1
timer2	16	0	0 to 65535	background processing timer 2
timer3	16	0	0 to 65535	background processing timer 3
timer4	16	0	0 to 65535	background processing timer 4
servo1	8	0	-100 to +100	servo motor 1 position
servo2	8	0	-100 to +100	servo motor 2 position
servo3	8	0	-100 to +100	servo motor 3 position
servo4	8	0	-100 to +100	servo motor 4 position
servo5	8	0	-100 to +100	servo motor 5 position
servo6	8	0	-100 to +100	servo motor 6 position
servo7	8	0	-100 to +100	servo motor 7 position
servo8	8	0	-100 to +100	servo motor 8 position
servo9	8	0	-100 to +100	servo motor 9 position

servo10	8	0	-100 to +100	servo motor 10 position
servo11	8	0	-100 to +100	servo motor 11 position
stepper1	16	0	-32768 to 32767	stepper motor position
motor1	8	0	-100 to +100	DC motor 1 speed
motor2	8	0	-100 to +100	DC motor 2 speed
error	8	0	0 or negative	non-zero if error occurred
ticks	16	0	0 to 65535	increments every 1/1000s (ms)
seconds	16	0	0 to 65535	increments every 1s
rand	16	?	0 to 65535	last random number
spi_in	1	0	0 or 1	set if SPI byte received
spi_out	1	1	0 or 1	set if SPI ready for next byte to be sent
uart_in	1	0	0 or 1	set if UART byte received
uart_out	1	1	0 or 1	set if UART ready for next byte to be sent
usb_in	1	0	0 or 1	set if USB byte received
usb_out	1	1	0 or 1	set if USB ready for next byte to be sent
usb_power	1	?	0 or 1	set if USB power present
stack	8	0	0 to 31	hardware stack pointer

Notes
1: The onboard LED is shared with pin 1. It is configured as an output by default.
2: The onboard pushbutton is only available to programs if 'Master Clear' is unchecked in PIC Options. Otherwise it functions as a reset button.

Appendix B - Predefined Symbols

Symbols which are predefined by the compiler. The use of them can make a program more readable.

Symbol	Value	Notes
off	0	off
on	1	on
lo	0	low level
hi	1	high level
false	0	false
true	1	true
pullup	1	weak pull-up on
none	0	no parity
even	1	even parity
odd	2	odd parity
standard	1	standard stepper sequence
high_torque	2	high-torque stepper sequence
half_step	3	half-step stepper sequence

Appendix C - Run-time Errors

Values that the variable *error* can take after an error has occurred (with 0 indicating no error).

Value	Error
-1	UART error (not configured)
-2	UART framing error
-3	UART parity error
-4	USB/UART overrun error
-5	SPI error (not configured)

Appendix D - Assembler Macros

The following macros are used to make assembler listings more readable.

Macro	Equivalent to
clrw	andlw 0
tstw	iorlw 0
comw	xorlw h'ff'
negw	sublw 0
movfw <reg>	movf <reg>,w
tstf <reg>	movf <reg>,f
clrc	bcf STATUS,C
setc	bsf STATUS,C
clrz	bcf STATUS,Z
setz	bsf STATUS,Z
skpc	btfss STATUS,C
skpnc	btfsc STATUS,C
skpz	btfss STATUS,Z
skpnz	btfsc STATUS,Z

Appendix E - Correspondence between JellyBean pins and PIC 18F14K50 pins

JellyBean	PIC
pin1	RC1/AN5/C12IN1-/INT1/VREF-
pin2	RC2/AN6/P1D/C12IN2-/CVREF/INT2
pin3	RB4/AN10/SDI/SDA
pin4	RB6/SCK/SCL
pin5	RB5/AN11/RX/DT
pin6	RC7/AN9/SDO/T1OSCO
pin7	RB7/TX/CK
pin8	RC6/AN8/SS/T13CKI/T1OSCI
pin9	RC3/AN7/P1C/C12IN3-/PGM
pin10	RC4/P1B/C12OUT/SRQ
pin11	RC5/CCP1/P1A/T0CKI
pin12	RA3/MCLR/VPP
led	shared with pin1
button	shared with pin12

Appendix F - Compiler Warning & Error Messages

If you encounter an error then examine the assembler code produced by the compiler (see Compiler Options for how to generate assembler code). This will indicate the context of the error. Errors are also listed in the Compilation Results window along with their line numbers. Double clicking an error message in this window will display the source line of the error. Pressing F1 while the cursor is within an error message in the Compilation Results window will display help on that error. Clicking the arrows at the bottom of the application window will step to the next or previous error.

Here is a complete list of compiler warning & error messages and their causes & resolutions:

Warnings

`Master Clear enabled – pin not available (#1)`	Disable Master Clear in PIC Options.
`Pin not configured as INPUT (#2)`	Configure pin as input.
`Timers and Interrupts option not enabled (#3)`	Enable option in Compiler Options.

Errors

`Label not found (#101)`	Branch or subroutine call to a non-existent label.
`Undefined variable – use DIM (#102)`	A variable has been used before being DIM'ed.
`Illegal variable or symbol name (#103)`	Name is a reserved keyword.
`Label defined twice (#104)`	Every label must be unique.
`Variable defined twice (#105)`	A variable has been DIM'ed after its first use. Move DIM statements to the start of the program.
`Too many labels (#106)`	Maximum of 32 labels in an ON statement.
`Unmatched FOR (#107)`	No corresponding NEXT statement.
`Unmatched NEXT (#108)`	No corresponding FOR statement.
`Multiple NEXT (#109)`	Mismatched FOR-NEXT loops.
`Too many FOR-NEXT loops (#110)`	Maximum of 256 loops in a program.
`Invalid pin (#111)`	Must be a number between 1 and 11, or 1 and 12.
`Invalid address (#112)`	Must be in the range 0 to 239.
`Invalid size (#113)`	Variable size must be 1, 8 or 16 bits.
`Syntax error (#114)`	Typically misspelled BASIC keyword, or incorrect arguments used.
`Invalid operation (#115)`	Operation not available.
`Number too big (#116)`	Greater than 255 or less than -256 for example.
`Shift too big (#117)`	Must be between 0 and 16.
`Not a constant expression (#118)`	Expression must not contain any variables.
`Divide by zero (#119)`	Attempt to divide by (constant) zero.
`Floating point not supported (#120)`	Attempt to use a number containing a decimal point.
`Weak pull-up not available (#121)`	Weak pull-ups only available on certain pins.
`Not analog pin (#122)`	Analog function only available on certain pins.
`Invalid UART format (#123)`	Use I/O Wizard to configure UART.
`Invalid SPI format (#124)`	Use I/O Wizard to configure SPI.
`SPI multiple receive (#125)`	Receive single byte at a time.
`LCD not configured (#126)`	Configure LCD before first use.
`SPI not configured (#127)`	Configure SPI before first use.
`UART not configured (#128)`	Configure UART before first use.

`Pin already in use (#129)`	Use a different pin.

Fatal errors

`Fatal error (#200)`	Not normally encountered.
`Internal error (#201)`	Not normally encountered.
`End of file (#202)`	Not normally encountered.
`Symbol table overflow (#203)`	Maximum of 4096 symbols, variables or labels.
`Too many variables (#204)`	RAM memory is full. Make variables smaller or double up their use.
`Program memory full (#205)`	Program memory is full. Make 16-bit variables into 8-bit unsigned variables.
`Out of memory (#206)`	Host PC error. Close some applications.

There are a couple of other errors that are not trapped at compile time. These fatal run-time errors are division by zero (other than division by the constant zero) and stack overflow/underflow. Both these conditions are indicated by the LED on the board flashing asymmetrically (on for longer than off).

Stack overflow occurs if subroutines are nested too deeply. The hardware stack is 31 levels deep but a number of these levels are required by the operating system. The exact stack space available to user programs depends on the specific peripherals in use but as a rule of thumb subroutines should not be nested more than 16 levels (each **GOSUB** instruction uses one level).

A stack underflow would typically indicate an unmatched **RETURN** instruction.

Run-time errors must be cleared by disconnecting the USB cable and any external power cable, closing the application then reconnecting the USB cable. As a precaution any program that has caused a fatal error is also erased from memory so must be downloaded again after correcting.