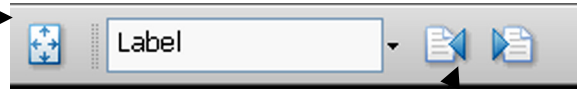


BASIC COMMANDS

This document provides the syntax (proper codes and usage) for all the BASIC commands supported by the AXEpad[®] for Snap Circuits[®] XP Editor and the PICAXE[®] system. It is intended as a lookup reference guide for each BASIC command that will be downloaded into the PICAXE[®] 08M2 microcontroller in the Snap Circuits[®] XP system.

Searching

Words in this document can be found and highlighted by entering the word in the box at the top of the screen that looks like this —→ and pressing <Enter> key. To find the next instance of that word click on forward arrow to the right of the box. To find previous instance click on back arrow. In this example we are searching for the word “Label”. You may also use the Bookmarks on the left to search for specific topics, key words, or chapters.



PICAXE[®] Software and Snap Circuits[®] XP components

The main application used for programming the PICAXE08M2 microcontroller is called the AXEpad for Snap Circuits[®] XP Editor (hereafter AXEpad for SCXP). There is more advanced programming software available free of charge to PICAXE[®] users at www.picaxe.co.uk

If you have a question about any command please post a question on the very active support forum at this website www.picaxeforum.co.uk

If you have questions about Snap Circuits[®] XP components go to www.snapcircuits.com or call Elenco at 847-541-3800.

UNDERSTANDING BASIC PROGRAMS

Labels

Labels are used as markers throughout a program. A ‘jump to’ is performed by using a *goto*, *gosub*, or other command. Labels can be any word (that is not a BASIC keyword) and may contain digits and the underscore character. Labels must start with a letter or underscore (not a digit), and are defined by ending with a colon (:).

The colon is not required in the *goto* or *gosub* commands.

The compiler used by AXEpad SCXP is not case sensitive (lower and/or upper case may be used at any time).

Example:

main:

High 4

Pause 500

Low 4

Pause 2000

GoTo main

‘this is the label “main”

‘ make pin 4 an output and switch it to B+.

‘ wait 1/2 second

‘ switch pin 4 to 0 volts or ground

‘ wait 2 seconds

‘ jump to the label “main:”

Whitespace

Whitespace is the term used to define the blank area on a printout of the program. If your printing on blue paper, this area of course would be a blue space. This involves spaces, tabs and empty lines. Any of these features can be used to space the program to make it clearer and easier to read. It is convention to only place labels on the left hand side of the screen. All other commands should be indented by using the 'tab key'. This convention makes the program much easier to read and follow.

Newline

Commands are normally placed on separate lines. However if desired the colon (:) character can be used to separate multiple commands on a single line as shown here,

```
if pin1 = 1 then : high 1 : else : low 1 : endif
```

Comments

Comments are used to add information into the program for future reference. They are completely ignored by the computer during a download. Comments begin with an apostrophe (') or semicolon (;) and continue until the end of the line. The keyword REM may also be used for a comment.

Multiple lines can be commented by use of the #REM and #ENDREM directives.

Examples:

high 0	' make output 0 high
low 0	REM make output 0 low
#rem	
high 0	
pause 1000	
low 0	
#endrem	

} ALL THESE WILL BE IGNORED DURING A DOWNLOAD

Constants

Constants are 'fixed' numbers that are used within the program. The software supports word integers (any whole number between 0 and 65535). Constants can be declared in four ways: decimal, hex, binary, and ASCII. Decimal numbers are typed directly without any prefix. Hexadecimal (hex) numbers are preceded with a dollar-sign (\$) or (0x).

Binary numbers are preceded by a percent-sign (%).

ASCII text strings are enclosed in quotes (").

Examples:

100	' 100 decimal
\$64	' 64 hex
0x64	' 64 hex
%01100100	' 01100100 binary
"A"	' "A" ascii (65)
"Hello"	' "Hello" - equivalent to "H", "e", "l", "l", "o"

Symbols

Symbols can be assigned to constant values, and can also be used as alias names for variables (see Variables overleaf for more details). Constant values and variable names are assigned by following the symbol name with an equal-sign (=), followed by the variable or constant. Symbols can use any word that is not a reserved keyword (for example ... *switch*, *step*, *output*, *input*, etc. **cannot be used**). Symbols can contain numeric characters and underscores (*flash1*, *flash_2* etc. **are acceptable**) but the first character cannot be a numeric (*1flash* **is not allowed**). Simple constant mathematics is also available. See the symbol command entry later in this document for more information.

Use of symbol does not increase program length.

Example:

'Define Symbols

symbol RED_LED = 4	' define a constant symbol
symbol COUNTER = b0	' define a variable symbol
let COUNTER = 200	' preload variable with value 200

'Program Using Above Symbols

mainloop:	' define a program address, address symbols end with colons
high RED_LED	' switch on output 4
pause COUNTER	' wait 0.2 seconds
low RED_LED	' switch off output 4
pause COUNTER	' wait 0.2 seconds
goto mainloop	' jump to the mainloop address symbol

Directives

Directives are used by the software to determine which sections of the program listing are to be compiled. Directives are therefore not part of the program, they are instructions to the software compiler. All directives start with a # and must be used on a single line. Any other nonrelevant line content after the directive is ignored.

#rem / #endrem

Comment out multiple lines of code.

Example:

#rem	'this code stops the compiling of code
high 0	'this line will not be compiled
pause 1000	'this line will not be compiled
low 0	'this line will not be compiled
#endrem	'this code starts the compiling of code

#include "filename"

Include code from a separately saved file within this program.

Example: **#include "c:\test.bas"**

Variables - General

The RAM memory is used to store temporary data in variables as the program runs. It

loses all data when the power is removed or reset. There are four types of RAM variables - general purpose, scratchpad, storage, and special function. See the 'let' command for details about variable mathematics.

General Purpose Variables.

Bytes, Bit Name, Byte Name, & Word Name

There are 14 general purpose byte variables. These byte variables are labelled b0, b1 etc... Byte variables can store integer numbers between 0 and 255. Byte variables cannot use negative numbers or fractions, and will 'overflow' without warning if you exceed the 0 or 255 boundary values (e.g. $254 + 3 = 1$) ($2 - 3 = 255$) However for larger numbers two byte variables can be combined to create a word variable, which is capable of storing integer numbers between 0 and 65,535. These word variables are labelled w0, w1, w2 etc... and are constructed as follows:

w0 = b1 : b0

w1 = b3 : b2

w2 = b5 : b4

w3 = b7 : b6

etc...

Therefore the most significant byte of w0 is b1, and the least significant byte of w0 is b0. In addition there are 16 individual bit variables (bit0, bit1 etc...). These bit variables can be used where you just require a single bit (0 or 1) storage capability. Bit variables are part of the lower value byte variables e.g.

b0 = bit7: bit6: bit5: bit4: bit3: bit2: bit1: bit0

b1 = bit15: bit14: bit13: bit12: bit11: bit10: bit9: bit8

etc...

You can use any word, byte or bit variable within any mathematical assignment or command that supports variables. **However take care that you do not accidentally repeatedly use the same 'byte' or 'bit' variable that is being used as part of a 'word' variable elsewhere.**

Variables - Special function

The special function variables available for the PICAXE® 08M2 are:

pins = the input / output port

dirs = the data direction register (sets whether pins are inputs or outputs)

infra = another term for variable b13, used within the 08M2 infrain2 command.

The variable pins is broken down into individual bit variables for reading from individual inputs with an if...then command. Only valid input pins are implemented.

pins = x : x : x : pin4 : pin3 : pin2 : pin1 : x

The variable dirs is also broken down into individual bits. Only valid bi-directional pin configuration bits are implemented.

dirs = x : x : x : dir4 : x : dir2 : dir1 : x

Variables - Mathematics

The PICAXE® 08M2 micro-controller supports word (16 bit) mathematics. Valid integers are 0 to 65535. All internal mathematics is 16 bit, however when, for instance, the

output target is a byte (8 bit) variable (0-255), if the result of the internal calculation is greater than 255 overflow will occur without warning. Math is performed strictly from left to right. Unlike some computers and calculators, the PICAXE® 08M2 does not give * and / priority over + and -. All mathematics is performed strictly from left to right.

Therefore 3+4x5 is calculated as:

3+4=7

7x5=35

The microcontroller does not support fractions or negative numbers. However it is sometimes possible to rewrite equations to use integers instead of fractions, e.g.

let w1 = w2 / 5.7 is not valid, but

let w1 = w2 * 10 / 57 is mathematically equal and valid.

The mathematical functions supported are:

+ ; add

- ; subtract

* ; multiply (returns low word of result)

** ; multiply (returns high word of result)

/ ; divide (returns quotient)

// % ; modulus divide (returns remainder)

MAX ; limit value to a maximum value

MIN ; limit value to a minimum value

AND & ; bitwise AND

OR | ; bitwise OR

XOR ^ ; bitwise XOR

NAND ; bitwise NAND

NOR ; bitwise NOR

XNOR ^/ ; bitwise XNOR

ANDNOT &/ ; bitwise AND NOT (NB this is *not* the same as NAND)

ORNOT |/ ; bitwise OR NOT (NB this is *not* the same as NOR)

On the PICAXE® 08M2 microcontroller it is not possible to enclose part equations in brackets for example let w1 = w2 / (b5 + 2) is not valid.

This would need to be entered in equivalent form as follows;

let w1 = b5 + 2

let w1 = w2 / w1

Further Information:

Addition and Subtraction

The addition (+) and subtraction (-) commands work as expected. Note that the variables will overflow without warning if the maximum or minimum value is exceeded (0-255 for bytes variables, 0-65535 for word variables).

Multiplication and Division

When multiplying two 16 bit word numbers the result is a 32 bit (double word) number. The multiplication (*) command returns the low word of a word*word calculation. The ** command returns the high word of the calculation and */ returns the middle word.

Therefore in normal maths \$aabb x \$ccdd = \$eeffgghh

In PICAXE maths

\$aabb * \$ccdd = \$gghh

\$aabb ** \$ccdd = \$eeff

The division (/) command returns the quotient (whole number) word of a word*word division. The modulus (// or %) command returns the remainder of the calculation.

MAX and MIN

The MAX command is a limiting factor, which ensures that a value never exceeds a preset value. In this example the value never exceeds 50. When the result of the multiplication exceeds 50 the max command limits the value to 50.

let b1 = b2 * 10 MAX 50

if b2 = 3 then b1 = 30

if b2 = 4 then b1 = 40

if b2 = 5 then b1 = 50

if b2 = 6 then b1 = 50 ' limited to 50

The MIN command is a similar limiting factor, which ensures that a value is never less than a preset value. In this example the value is never less than 50. When the result of the division is less than 50 the min command limits the value to 50.

let b1 = 100 / b2 MIN 50

if b2 = 1 then b1 = 100

if b2 = 2 then b1 = 50

if b2 = 3 then b1 = 50 ' limited to 50

MORE BASIC COMMANDS

AND, OR, XOR, NAND, NOR, XNOR, ANDNOT, ORNOT

The AND, OR, XOR, NAND, NOR, XNOR commands function bitwise on each bit in the variables. ANDNOT and ORNOT mean, for example 'A AND the NOT of B' etc. This is not the same as NOT (A AND B), as with the traditional NAND command.

A common use of the AND (&) command is to mask individual bits:

let b1 = pins & %00000110

This masks inputs 1 and 2, so the variable only contains the data of these two inputs.

<< , >>

Shift left (or shift right) have the same effect as multiplying (or dividing) by 2. All bits in the word are shifted left (or right) a number of times. The bit that 'falls off' the left (or right) side of the word is lost.

let b1 = %00000110 << 2

DIG

The DIG (digit) command returns the decimal value of a specified digit (0-4, right to left) of a 16 bit number. Therefore digit 0 of '67890' is 0 and digit 3 is '7'. To return the ASCII value of the digit simply add string "0" to the digit value for example;

let b1 = b2 DIG 0 + "0"

See also the BINTOASCII and BCDTOASCII commands.

REV

The REV (reverse) command reverses the order of the specified number of bits of a 16 bit number. Therefore to reverse the 8 bits of %10110000 (to %00001101) the command would be;

```
let b1 = %10110000 REV 8
```

Variables - Unary Mathematics

The microcontroller supports the NOT unary command, for example;

```
let b1 = NOT pins
```

Further Information:

NOT

The NOT function inverts a value.

For example,

```
let b1 = NOT %01110000
```

 (answer b1 = %10001111)

bcdtoascii:

Syntax:

BCDTOASCII variable, tens, units

BCDTOASCII wordvariable, thousands, hundreds, tens, units

- Variable contains the value (0-99) or wordvariable (0-9999)
- Thousands receives the ASCII value ("0" to "9")
- Hundreds receives the ASCII value ("0" to "9")
- Tens receives the ASCII value ("0" to "9")
- Units receives the ASCII value ("0" to "9")

Function:

Convert a BCD value into separate ASCII bytes.

Information:

This is a 'pseudo' command designed to simplify the conversion of byte or word BCD values into ASCII. Note that the maximum valid value for a BCD value is 99 (byte) or 9999 (word).

Example:

```
main: inc b1
bcdtoascii b1,b2,b3      ' convert to ascii
debug                   ' debug values for testing
goto main               ' loop back to start
```

bintoascii:

Syntax:

BINTOASCII variable, hundreds, tens, units

BINTOASCII wordvariable, tenthousands, thousands, hundreds, tens, units

- Variable contains the value (0-255) or wordvariable (0-65535)
- TenThousands receives the ASCII value ("0" to "9")
- Thousands receives the ASCII value ("0" to "9")
- Hundreds receives the ASCII value ("0" to "9")
- Tens receives the ASCII value ("0" to "9")

- Units receives the ASCII value ("0" to "9")

Function:

Convert a binary value into separate ASCII bytes.

Information:

This is a 'pseudo' command designed to simplify the conversion of byte or word binary values into ASCII. *Example:*

main:

inc b1

bintoascii b1,b2,b3,b4 ' convert b1 to ascii

debug ' debug values for testing

goto main ' jump to main label

branch:

Syntax:

BRANCH offset,(address0,address1...addressN)

- Offset is a variable/constant which specifies which Address# to use (0-N).

- Addresses are labels which specify where to go.

Function:

Branch to address specified by offset (if in range).

Information:

This command allows a jump to different program positions depending on the value of the variable 'offset'. If offset is value 0, the program flow will jump to address0, if offset is value 1 program flow will jump to address1 etc. If offset is larger than the number of addresses the whole command is ignored and the program continues at the next line. This command is identical in operation to on ... goto

Example; Use circuit shown here with this program (BRANCH_demo.bas).

reset1:

let b1 = 255 ' When b1=255 an inc command overflows b1 to 0

low 0 ' First LED OFF

low 4 ' Second LED OFF

 ' After initial conditions are set start main program

main: inc b1 ' After a reset 'inc b1' makes b1 = 0

pause 1000 ' Wait for 1 second

if b1 > 4 then reset1

branch b1, (btn0,btn1, btn2, btn3, btn4) ' Branch Command

btn0: high 0 ' First LED ON

goto main

btn1: low 0 ' First LED OFF

goto main

btn2: high 4 ' Second LED ON

goto main

btn3: low 4 ' Second LED OFF

goto main

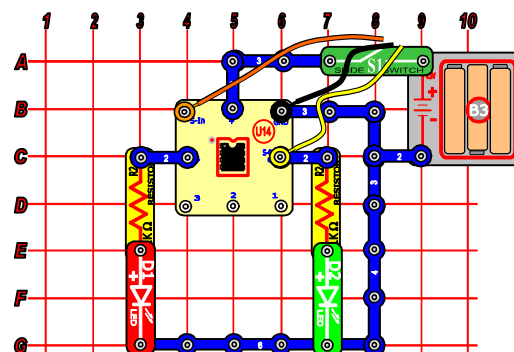
btn4: high 4 ' Both LED's ON

high 0

goto main

button:

Syntax:



BUTTON pin,downstate,delay,rate,bytevariable,targetstate,address

- Pin is a variable/constant (1-4) which specifies the i/o pin to use.
- Downstate is a variable/constant (0 or 1) which specifies what logical state is read when the button is pressed.
- Delay is a variable/constant (0-254) which is a counter which specifies number of loops before a repeat if BUTTON is used within a loop. A value of 255 disables this feature.
- Rate is a variable/constant (0-255) which specifies the auto-repeat rate in BUTTON cycles.
- Bytevariable is the workspace. It must be cleared to 0 before being used by BUTTON for the first time (before the loop that BUTTON is in)
- Targetstate is a variable/constant (0 or 1) which specifies what state (0=not pressed, 1=pressed) the button should be in for a branch to occur.
- Address is a label which specifies where to go if the button is in the target state.

Function:

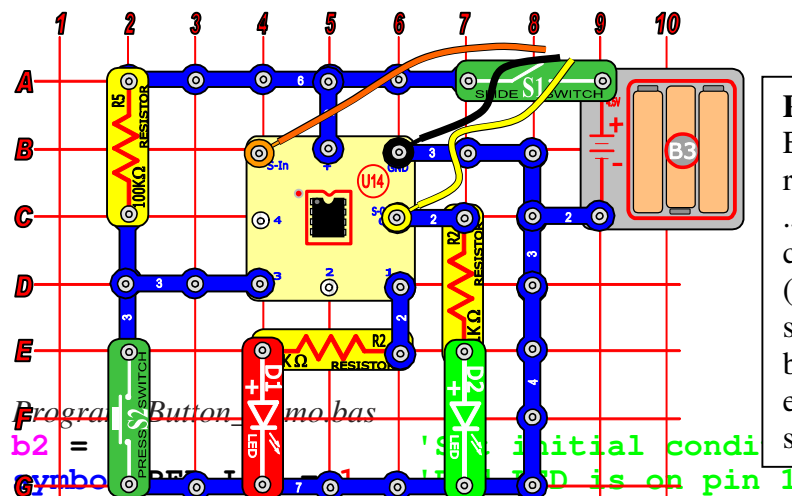
Debounce button, auto-repeat, and branch if button is in target state.

Information:

When mechanical switches are activated the metal ‘contacts’ do not actually close in one smooth action, but ‘bounce’ against each other a number of times before settling. This is called electrical noise and the microcontroller can register each bounce as a new button closure. One way of overcoming this is to add a small pause within the program to give time for the switch to settle. The button command can also be used to overcome switch noise. When the button command is executed, the microcontroller looks to see if the ‘downstate’ is matched. If this is true the switch is debounced, and then program flow jumps to ‘address’ if ‘targetstate’ = 1. If targetstate = ‘0’ the program continues. If the button command is within a loop, the next time the command is executed ‘downstate’ is once again checked. If the condition is still true, the variable ‘bytevariable’ is incremented. This can happen a number of times until ‘bytevariable’ value is equal to ‘delay’. At this point a jump to ‘address’ is made if ‘targetstate’ = 1. Bytevariable is then reset to 0 and the whole process then repeats, but this time the jump to ‘address’ is made when the ‘bytevariable’ value is equal to ‘rate’. This gives action like a computer keyboard key press - send one press, wait for ‘delay’ number of loops, then send multiple presses at time interval ‘rate’.

Note that button should be used within a loop. It does not pause program flow and so only checks the input switch condition as program flow passes through the command.

Example; Use this circuit and program to test button command.



PROGRAM DESCRIPTION

Every time the S2 button is pressed the red LED will come on and stay on for .5 seconds times the number of S2 clicks. If S2 is held for 21 button cycles (20 green flashes) the auto-repeat will start and send a command every 6 button cycles (5 green flashes). After each send the length that the red LED stays on is increased by .5 seconds.

```

symbol GREEN_LED = 0 'Green LED is on pin 0
w2 = 0 'End of initial conditions and symbols
main: 'IF BUTTON HELD FOR 20 GREEN FLASHES, REPEAT SENT ON EVERY 5TH FLASH
button 3,0,21,6,b2,1,cont 'jump to cont when button is pressed
high GREEN_LED 'if no button pressed then turn on green LED
pause 250 'pause .25 seconds
low GREEN_LED 'turn off green LED
pause 250 'pause .25 seconds
goto main 'repeat button cycle, approximately .5 seconds/cycle
cont:
low GREEN_LED 'Turn off green LED
high RED_LED 'Turn on red LED
w2 = w2 + 500 'increase delay by .5 seconds
pause w2 'pause for delay = button presses times .5 seconds
low RED_LED 'turn off red LED
goto main 'repeat process

```

calibfreq: NOT RECOMMENDED FOR BEGINNERS

Syntax:

CALIBFREQ {-} factor

- factor is a constant/variable containing the value -15 to 15

Function:

Calibrate the microcontrollers internal resonator. 0 is the default factory setting.

Information:

The microcontroller used by the Snap Circuit XP product has an internal resonator that can be set to 4 or 8MHz operation via the setfreq command. It is also possible to ‘calibrate’ this frequency’ of this microcontroller. This is an advanced feature not normally required by most users. Generally the only use for calibfreq is to slightly adjust the frequency for serial transactions with third party devices. A larger positive value increases speed, a larger negative value decreases speed. Try the values -4 to + 4 first, before going to a higher or lower values.

Use this command with extreme care. It can alter the frequency of the PICAXE chip beyond the serial download tolerance - in this case you will need to perform a ‘hard-reset’ in order to carry out a new download.

count:

Syntax:

COUNT pin, period, wordvariable

- Pin is a variable/constant (0-4) which specifies the input pin to use.

- Period is a variable/constant (1-65535ms at 4MHz).

- Wordvariable receives the result (0-65535).

Function:

Count pulses on an input pin.

Information:

Count checks the state of the input pin and counts the number of low to high transitions within the time ‘period’. A word variable should be used for ‘variable’. At 4MHz the input pin is checked every 20us (micro-seconds or .00002 seconds), so the highest frequency of pulses that can be counted is 25kHz (kilo-hertz or Cycles/second), presuming a 50% duty cycle (equal high-low time). Take care with mechanical switches, which may cause multiple ‘hits’ for each switch

push as the metal contacts 'bounce' upon closure.

Affect of increased clock speed:

The period value is 0.5ms at 8MHz and 0.25ms at 16MHz.

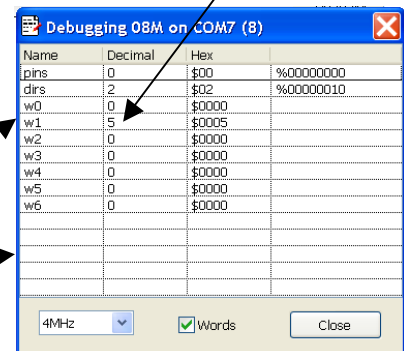
At 8MHz the input pin is checked every 10us, so the highest frequency of pulses that can be counted is 50kHz, presuming a 50% duty cycle (equal high-low time). At 16MHz the input pin is checked every 5us, so the highest frequency of pulses that can be counted is 100kHz, presuming a 50% duty cycle (equal high-low time).

Example: Use circuit for button: Download COUNT_Demo.bas. Press S2 & count bounces.

```
main:
high 1          ' turn on red LED
count 3, 5000, w1 ' count pulses in 5 seconds
low 1           ' turn off red LED
low 0           ' recommended before a debug
pause 500       ' allow things to settle
debug w1        ' display value
pause 3000      ' allow 8 (3+5) seconds to read w1
w1 = 0          ' clear counter
goto main       ' jump to main
```

Note: Replace S1 with a 3 space and put S1 in button circuit instead of S2 to check the noise in a slide switch. One closing produced the 5 readings shown in w1.

SWITCH NOISE CAN CAUSE ERRORS IN MANY CIRCUITS!



Name	Decimal	Hex	
pins	0	\$00	%00000000
dhrs	2	\$02	%00000010
w0	0	\$0000	
w1	5	\$0005	
w2	0	\$0000	
w3	0	\$0000	
w4	0	\$0000	
w5	0	\$0000	
w6	0	\$0000	

debug:

Syntax:

DEBUG {var}

-var is an optional variable value (for example b1). It's value is not of importance and is included purely for compatibility with older programs.

Function:

Display variable information in the debug window when the debug command is processed. Byte information is shown in decimal, binary, hex and ASCII notation. Word information is shown in decimal and hex notation when word box is checked.

Information:

The debug command uploads the current variable values for ALL the variables via the download cable to the computer screen. This enables the computer screen to display all the variable values in the microcontroller for debugging purposes. Note that the debug command uploads a large amount of data and so significantly slows down any program loop. To display user defined debugging messages use the 'sertxd' command instead. Note that on the 08M2 microcontroller, debug acts on 'output 0'. Therefore programs that use output 0 may corrupt the serial data condition. In this case it is recommended to use the following structure before a debug command.

```
low 0          ' reset 0 to correct condition
pause 500      ' wait a while
debug b1       ' display values on computer screen
```

Affect of increased clock speed:

When using an 8MHz clock speed ensure the software has been set with the correct speed setting to enable successful communication between microcontroller and PC.

Example: Type in this program to check communication to computer

```
main:  
let b1 = b1 + 1  \ increment value of b1  
readadc 2,b2    \ read an analogue value  
debug b1        \ display values on computer screen  
pause 500       \ wait 0.5 seconds  
goto main       \ loop back to start
```

dec:

Syntax:

DEC var

- var is the variable to decrement

Function:

Decrement (subtract 1 from) the variable value.

Information:

This command is shorthand for 'let var = var - 1'

Example:

```
for b1 = 1 to 5  
dec b2  
next b1
```

disablebod:

Syntax:

DISABLEBOD

Function:

Disable the on-chip brown out detect function.

Information:

Some PICAXE chips have a programmable internal brown out detect function, to automatically cleanly reset the chip on a power brown out. The brown out detect is always enabled by default when a program runs. However it is sometimes beneficial to disable this function to reduce current drain in battery powered applications whilst the chip is 'sleeping'. Use of the disablebod command prior to a sleep will considerably reduce the current drawn during the actual sleep command.

Example:

```
main: disablebod \ disable brown out  
sleep 10         \ sleep for 23 seconds  
enablebod       \ enable brown out  
goto main        \ loop back to start
```

do...loop:

Syntax:

DO

{code}

LOOP UNTIL/WHILE VAR ?? COND

DO

```
{code}  
LOOP UNTIL/WHILE VAR ?? COND AND/OR VAR ?? COND...  
DO UNTIL/WHILE VAR ?? COND  
{code}  
LOOP  
DO UNTIL/WHILE VAR ?? COND AND/OR VAR ?? COND...  
{code}  
LOOP
```

- var is the variable to test

- cond is the condition

?? can be any of the following conditions;

= equal to	is equal to
<> not equal to	!= not equal to
> greater than	< less than

Function:

Loop whilst a condition is true (while) or false (until)

Information:

This structure creates a loop that allows code to be repeated whilst, or until, a certain condition is met. The condition may be in the 'do' line (condition is tested before code is executed) or in the 'loop' line (condition is tested after the code is executed).

The exit command can be used to prematurely exit out of the do...loop.

Example: Use button circuit for this program (DO_LOOP_Demo.bas).

start:

```
b1 = 0                ' set initial conditions  
low 0  
high 1               ' last initial condition  
do                   ' start DO LOOP  
    toggle 0  
    pause 250  
    toggle 1  
    pause 250  
    inc b1  
    if pin3 = 0 then exit    ' allow manual exscape  
loop while b1 < 10         ' end after 10 cycles  
low 1  
pause 2000
```

needS2:

```
if pin3 = 0 then start    ' repeat if button pressed  
goto needS2
```

eprom: (data)

Syntax:

DATA {location},(data,data...)

EEPROM {location},(data,data...)

- Location is an optional constant (0-255) which specifies where to begin storing the data in the EEPROM. If no location is specified, storage continues from where it last left off. If no location was initially specified, storage begins at 0.

Data are constants (0-255) which will be stored in the EEPROM.

Function:

Preload EEPROM data memory. If no EEPROM command is used the values are automatically cleared to the value 0. The keywords DATA and EEPROM have identical functions and either can be used.

Information:

This is not an instruction, but a method of pre-loading the microcontroller data memory. The command does not affect program length. With the PICAXE-08M the data memory is shared with program memory. Therefore only unused bytes may be used within a program. To establish the length of the program use 'Check Syntax' from the PICAXE menu. This will report the length of program. Available data addresses can then be used as follows:

PICAXE-08M 0 to (255 - number of used bytes)

Example: Use button circuit, download EEPROM_Demo.bas, then press F8 to display terminal

```
EEPROM 0, ( "Hello World",13,10 ) ' save message in EEPROM
main:
    pause 100                      ' let settle
    for b0 = 0 to 12                ' start a loop
        read b0,b1                 ' read value from EEPROM
        serout 0,N2400,(b1)         ' transmit to Terminal (F8)
    next b0                        ' next character
    pause 500                      ' let settle
needs2:
    if pin3 = 0 then main           ' if button pressed do again
    goto needs2                    ' else wait for button
```

enablebod:

Syntax:

ENABLEBOD

Function:

Enable the on-chip brown out detect function.

Information:

Some PICAXE chips have a programmable internal brown out detect function, to automatically cleanly reset the chip on a power brown out. The brown out detect is always enabled by default when a program runs. However it is sometimes beneficial to disable this function to reduce current drain in battery powered applications whilst the chip is 'sleeping'. Use of the disablebod command prior to a sleep will considerably reduce the current drawn during the actual sleep command.

Example:

```
main: disablebod ` disable brown out
sleep 10 ` sleep for 23 seconds
enablebod ` enable brown out
goto main ` loop back to start
```

end:

Syntax:

END

Function:

Sleep terminally until the power cycles (program re-runs) or the PC connects for a new download. Power is reduced to an absolute minimum (assuming no loads are being driven) and internal timers are switched off.

Information:

The end command places the microcontroller into low power mode after a program has finished. Note that as the compiler always places an END instruction after the last line of a program, this command is rarely required. The end command switches off internal timers, and so commands such as servo and pwmout that require these timers will not function after an end command has been completed. If you do not wish the end command to be carried out, place a 'stop' command at the bottom of the program. The stop command does not enter low power mode. The main use of the end command is to separate the main program loop from sub-procedures as in the example below. This ensures that programs do not accidentally 'fall into' the sub-procedure.

Example:

```
main:
let b2 = 15      \ set b2 value
pause 2000      \ wait for 2 seconds
gosub flsh      \ call sub-procedure
let b2 = 5      \ set b2 value
pause 2000      \ wait for 2 seconds
end             \ stop accidentally falling into sub
flsh:
for b0 = 1 to b2 \ define loop for b2 times
high 1          \ switch on output 1
pause 500       \ wait 0.5 seconds
low 1           \ switch off output 1
pause 500       \ wait 0.5 seconds
next b0         \ end of loop
return          \ return from sub-procedure
```

exit:

Syntax:

EXIT

Function:

Exit is used to immediately terminate a do...loop or for...next program loop.

Information:

The exit command immediately terminates a do...loop or for...next program loop. It is equivalent to 'goto line after end of loop'.

Example:

```
main:
do              \ start loop
if b1 = 1 then
exit
end if
loop           \ loop
```


for...next:

Syntax:

FOR variable = start TO end {STEP {-}increment}

(other program lines)

NEXT {variable}

- Variable will be used as the loop counter
- Start is the initial value of variable
- End is the finish value of variable
- Increment is an optional value which overrides the default counter value of +1. If Increment is preceded by a '-', it will be assumed that Start is greater than End, and therefore increment will be subtracted (rather than added) on each loop.

Function:

Repeat a section of code within a FOR-NEXT loop.

Information:

For...next loops are used to repeat a section of code a number of times. When a byte variable is used, the loop can be repeated up to 255 times. Every time the 'next' line is reached the value of variable is incremented (or decremented) by the step value (+1 by default). When the end value is exceeded the looping stops and program flow continues from the line after the next command. For...next loops can be nested 8 deep (remember to use a different variable for each loop). The for...next loop can be prematurely ended by use of the exit command.

Example: Use BUTTON circuit. LED flashes 20 times unless button is pushed.

main:

for b0 = 1 to 20	` define loop for 20 times
if pin3 = 0 then exit	` Push switch to exit
high 1	` switch on LED
pause 500	` wait 0.5 seconds
low 1	` switch off LED
pause 500	` wait 0.5 seconds
next b0	` end of loop
pause 9000	` wait for 9 seconds
goto main	` loop back to start

gosub:

Syntax:

GOSUB address

- Address is a label which specifies where to gosub to.

Function:

Go to sub procedure at 'address', then 'return' at a later point.

Information:

The gosub ('goto subprocedure') command is a 'temporary' jump to a separate section of code, from which you will later return (via the return command). Every gosub command **MUST** be matched by a corresponding return command. Do not confuse with the 'goto' command which is a permanent jump to a new program location. Gosubs can be nested 4 levels deep (there is a 4 level stack available in the microcontroller).

Sub procedures are commonly used to reduce program space usage by putting repeated sections of code in a single sub-procedure. By passing values to the subprocedure within variables, you

can repeat a section of code from multiple places within the program. See the sample below for more information.

Example: Use Button Circuit

```
main:
let b2 = 15      \ set b2 value
gosub flsh       \ call sub-procedure
let b2 = 5        \ set b2 value
gosub flsh       \ call sub-procedure
end              \ stop accidentally falling into sub

flsh:           \ sub-procedure
for b0 = 1 to b2 \ define loop for b2 times
high 1           \ switch on output 1
pause 500        \ wait 0.5 seconds
low 1            \ switch off output 1
pause 500        \ wait 0.5 seconds
next b0          \ end of loop
return          \ return from sub-procedure
```

goto:

Syntax:

GOTO address

- Address is a label which specifies where to go.

Function:

Go to address.

Information:

The goto command is a permanent 'jump' to a new section of the program. The jump is made to a label.

Example: Use Button Circuit. Turns red LED on for 5 seconds then off for 5 seconds.

```
main:
high 1           \ switch on output 1
pause 5000       \ wait 5 seconds
low 1            \ switch off output 1
pause 5000       \ wait 5 seconds
goto main        \ loop back to start
```

high:

Syntax:

HIGH pin {,pin,pin...}

- Pin is a variable/constant (0, 1, 2, or 4) which specifies the pin to use.

Function:

Make pin output high.

Information:

The high command switches an output on (high).

The PICAXE 08M2 microcontroller has configurable input/output pins so this command also automatically configures the pin as an output. This command will not work on pin 3 since it is always an input pin.

Example: Use Button Circuit. Turns red LED on for 5 seconds then off for 5 seconds.

```
main:
high 1          \ switch on output 1
pause 5000      \ wait 5 seconds
low 1           \ switch off output 1
pause 5000      \ wait 5 seconds
goto main       \ loop back to start
```

if...then: \ elseif...then: \ else: \ endif:

Syntax:

```
IF variable ?? value {AND/OR variable ?? value ...} THEN
{code}
ELSEIF variable ?? value {AND/OR variable ?? value ...} THEN
{code}
ELSE
{code}
ENDIF
```

- Variable(s) will be compared to value(s).
- Value is a variable or a constant.
- Bit is the bit number to check if set (1) or clear (0)

?? can be any of the following conditions

= equal to	is equal to
<> not equal to	!= not equal to
> greater than	>= greater than or equal to
< less than	<= less than or equal to

Function:

Compare and conditionally execute sections of code.

Information:

The multiple line if...then\ elseif \ else \ endif command is used to test input pin variables (or general variables) for certain conditions. If these conditions are met that section of the program code is executed, and then program flow jumps to the endif position. If the condition is not met program flows jumps directly to the next elseif or else command. The 'else' section of code is only executed if none of the if or elseif conditions have been true. When using inputs the input variable (pin1, pin2 etc) must be used (not the actual pin name 1, 2 etc.) i.e. the line must read 'if pin1 = 1 then...', not 'if 1 = 1 then...'

Note that

```
if b0 > 1 then (goto) label_1  \ (single line structure)
if b0 > 1 then gosub label_1    \ (single line structure)
if b0 > 1 then...else...endif   \ (multi line structure)
```

are 3 completely separate structures which cannot be combined. Therefore the following line is invalid as it tries to combine both a single and multi-line structure

```
if b0 > 1 then goto label_1 else goto label_2
```

This is invalid as the compiler does not know which structure you are trying to use

ie:

```
if b0 > 1 then goto label_1 : else : goto label_2
```

or

```
if b0 > 1 then : goto label_1 : else : goto label_2
```

To achieve this structure the line must be re-written as

```
if b0 > 1 then
goto label_1
else
goto label_2
endif
```

or

```
if b0 > 1 then : goto label_1 : else : goto label_2 : endif
```

The “:” character separates the sections into correct syntax for the compiler.

if...then: {goto}

if...and/or..then: {goto}

Syntax:

IF variable ?? value {AND/OR variable ?? value ...} THEN address

- Variable(s) will be compared to value(s).
- Value is a variable/constant.
- Address is a label which specifies where to go if condition is true.

The keyword goto after then is optional.

?? can be any of the following conditions

= equal to	is equal to
<> not equal to	!= not equal to
> greater than	>= greater than or equal to
< less than	<= less than or equal to

Function:

Compare and conditionally jump to a new program position.

Information:

The if...then command is used to test input pin variables (or general variables) for certain conditions. If these conditions are met program flow jumps to the new label. If the condition is not met the command is ignored and program flow continues on the next line. When using inputs the input variable (pin1, pin2 etc) must be used (not the actual pin name 1, 2 etc.) i.e. the line must read ‘if pin1 = 1 then...’, not ‘if 1 = 1 then...’ The if...then command only checks an input at the time the command is processed. Therefore it is normal to put the if...then command within a program loop that regularly scans the input. For details on how to permanently scan for an input condition using interrupts see the ‘setint’ command.

Example :Use Button Circuit. Toggles LED’s when button is pushed or held down

Checking an input within a loop. Wait 1 second after button is pressed or hold button down.

start:

```
High 1           \ turn on red LED
```

main:

```
if pin3 = 0 then flsh \ jump to flsh if pin3 is pressed
```

```
goto main          \ else loop back and look again
```

flsh:

```
toggle 1,0        \ change outputs 0,1
```

```
pause 1000         \ wait 1 second
```

```
goto main          \ loop back to main
```

if...and/or...then: exit:

IF variable ?? value {AND/OR variable ?? value ...} THEN EXIT
IF variable BIT value SET/CLEAR THEN EXIT (*X1/X2 parts only*)

- ?? can be any of the following conditions

$\langle \rangle$ not equal to $\neq$ not equal to

Compare and conditionally exit a do...loop or for...next loop

The if...then exit command is used to test input pin variables (or general variables) for certain conditions. If these conditions are met the current loop (do...loop or for...next) is prematurely ended. Multiple compares can be combined with the AND and OR keywords. For examples on how to use AND and OR see the if...then goto command.

Checking an input within a do loop and exit loop if condition is met.

```
High 1           ` turn on red LED
main:
do               ` start do loop
if pin3 = 0 then exit ` jump to flsh if pin3 is pressed
loop            ` else loop back and look again
flsh:
toggle 1,0      ` change outputs 0,1
pause 1000      ` wait 1 second
goto main       ` go back to main
```

if...and/or...then: gosub:

IF variable ?? value {AND/OR variable ?? value ...} THEN GOSUB address
IF variable BIT value SET/CLEAR THEN GOSUB address *(X1/X2 parts only)*

- ?? can be any of the following conditions

$\langle \rangle$ not equal to $\neq$ not equal to

> greater than >= greater than or equal to

< less than

<= less than or equal to

Function:

Compare and conditionally execute a gosub command.

Information:

The if...then gosub command is used to test input pin variables (or general variables) for certain conditions. If these conditions are met a sub procedure is executed. If the condition is not met the command is ignored and program flow continues on the next line. Any executed sub procedure returns to the next line. When using inputs the input variable (pin1, pin2 etc) must be used (not the actual pin name 1, 2 etc.) . For example the line must read 'if pin1 = 1 then gosub...', not 'if 1 = 1 then gosub...'. The if...then gosub command only checks an input at the time the command is processed. Therefore it is normal to put the if...then command within a program loop that regularly scans the input. Multiple compares can be combined with the AND and OR keywords. For examples on how to use AND and OR see the if...then goto command.

Example: Use Button Circuit: Turns OFF Green and Turns ON RED then Locks out user.

Checking an input within a loop.

```
start:
b0 = 1           \ lockout variable
high 0           \ set initial conditions
main:
if pin3 = 0 and b0 = 1 then gosub flsh \ jump if condition is met
goto main        \ else loop back and look again
flsh:
toggle 1,0       \ change outputs 0,1
b0 = 0
pause 1000       \ wait 1 second
return           \ go back and loop until program restarts
```

AND/OR Gates

2 input AND gate

```
if pin1 = 1 and pin2 = 1 then gosub label
```

3 input AND gate

```
if pin0 = 1 and pin1 = 1 and pin2 = 1 then gosub label
```

2 input OR gate

```
if pin1 = 1 or pin2 = 1 then gosub label
```

Analogue value between certain values

```
readadc 1,b1
```

```
if b1 >= 100 and b1 <= 200 then gosub label
```

To read the whole input port at once the variable 'pins' can be used

'jumps to label if inputs 0,1,2,3,& 4 are all high.

To read the whole input port and mask individual inputs (for example read only pins 2,3, & 4).

Example: Use Circuit Shown Below.

start:

```
high 1      'start with red LED on.
```

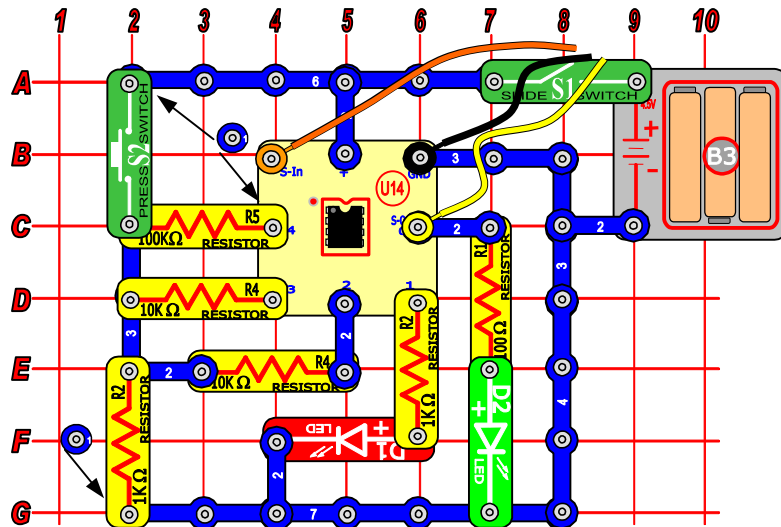
```
low 0       'start with green LED off.
```

main:

```

let b1 = pins & %00011100      'inputs on pins 2,3,4
if b1 = %00011100 then gosub label_1 'all high goto label_1
goto main                      'loop until all inputs are high
label_1:
toggle 0,1                    ' change state of each LED
pause 500                     ' wait for .5 seconds
goto main                     ' go back to beginning of main program

```



Shorting any one or two pins from the group 2,3,&4 to + will not produce a change. By pressing the pushbutton S2, all three pins are taken to + voltage and the LED's will now blink back an forth. This circuit is equivalent to a three input AND gate. Pin2 AND pin3

inc:

Syntax:

INC var

- var is the variable to increment

Function: Increment (add 1 to) the variable value.

Information: This command is shorthand for 'let var = var + 1'

Example:

```

for b1 = 1 to 5
inc b2
next b1

```

infrain2:

Syntax:

INFRAIN2

Function:

Wait until a new infrared command is received.

Description:

This command waits for an infrared signal from the infrared TV style transmitter or another PICAXE-08M source. All processing stops until the new command is received. The value of the command received is placed in the predefined variable 'infra'. This will be a number between 0 and 127. See the infraout command description for more details about the values that will be received from the TVR010 remote control. On the PICAXE-08M 'infra' is another name for 'b13' - it is the same variable. The infra-red input is fixed to a single input - see the PICAXE pinout diagrams. After using this command you may have to perform a 'hard reset' to download

a new.

Affect of Increased Clock Speed:

This command will only function at 4MHz. Use a setfreq m4 command before this command if using 8MHz speed,

Example:

```
main:
    infrain2                'wait for new signal
    if infra = 1 then swon1  'switch on 1
    if infra = 4 then swoff1 'switch off 1
    goto main
swon1:
    high 1
    goto main
swoff1:
    low 1
    goto main
```

infraout:

Syntax:

INFRAOUT device,data

- device is a constant/variable (valid device ID 1-31)
- data is a constant/variable (valid data 0-127)

Function:

Transmit an infra-red signal, modulated at 38kHz.

Description:

This command is used to transmit the infra-red data to Sony TM device (can also be used to transmit data to another PICAXE-08M that is using the infrain or infrain2 command). Data is transmitted via an infra-red LED (connected on output 0) using the SIRC (Sony Infra Red Control) protocol. For more information see PICAXE Manual Part 2 – Basic Commands.

input:

Syntax:

INPUT pin,pin,pin...

- Pin is a variable/constant which specifies the i/o pin to use.

Function:

Make pin an input.

Information:

This command can be used to change a pin that has been configured as an output back to an input. All pins are configured as inputs on first power-up (unless the pin is a fixed output). Fixed pins are not affected by this command. These pins are:

pin 0 = fixed output, pin 3 = fixed input

Example:

```
main:
input 1 ` make pin input
```

```
reverse 1 ` make pin output
reverse 1 ` make pin input
output 1 ` make pin output
```

let:

Syntax:

{LET} variable = {-} value ?? value ...

- Variable will be operated on.
- Value(s) are variables/constants which operate on variable.

Function:

Perform variable manipulation (wordsize-to-wordsize).

Maths is performed strictly from left to right.

The 'let' keyword is optional.

Information:

The microcontroller supports word (16 bit) mathematics. Valid integers are 0 to 65535. All mathematics can also be performed on byte (8 bit) variables (0-255). The microcontroller does not support fractions or negative numbers. However it is sometimes possible to rewrite equations to use integers instead of fractions, for example;

let w1 = w2 / 5.7 is not valid, but **let w1 = w2 * 10 / 57** is mathematically equal and valid.

The mathematical functions supported by all parts are:

+ ; add

- ; subtract

* ; multiply (returns low word of result)

** ; multiply (returns high word of result)

/ ; divide (returns quotient)

// (or %) ; modulus divide (returns remainder)

MAX ; limit value to a maximum value

MIN ; limit value to a minimum value

AND & ; bitwise AND

OR | ; bitwise OR (typed as SHIFT + \ on UK keyboard)

XOR ^ ; bitwise XOR (typed as SHIFT + 6 on UK keyboard)

NAND ; bitwise NAND

NOR ; bitwise NOR

ANDNOT &/ ; bitwise AND NOT (NB this is *not* the same as NAND)

ORNOT |/ ; bitwise OR NOT (NB this is *not* the same as NOR)

XNOR ^/ ; bitwise XOR NOT (same as XNOR)

The X1 and X2 parts also support

<< ; shift left

>> ; shift right

*/ ; multiply (returns middle word of result)

It is not possible to enclose part equations in brackets;

let w1 = w2 / (b5 + 2) Not valid.

This would need to be entered as:

let w1 = b5 + 2

let w1 = w2 / w1

let: dirs: =

Syntax:

{LET} dirs = value

- Value(s) are variables/constants which operate on the data direction register.

Function:

Configure pins as inputs or outputs.

Information:

The picaxe08m microcontroller allows some pins to be configured as inputs or outputs. These pins are configured as inputs on first power up. There are a number of ways to configure these pins:

- 1) Use the input/output/reverse commands.
- 2) Use an output command (high, pulsout etc) that automatically configures the pin as an output.
- 3) Use the let dirs = statement.

When working with this statement it is conventional to use binary notation. With binary notation pin 7 is on the left and pin 0 is on the right. If the bit is set to 0 the pin will be an input, if the bit is set to 1 the pin will be an output. Note that the 8 pin PICAXE has some pre-configured pins (e.g. pin 0 is always an output and pin 3 is always an input). Adjusting the bits for these pins will have no affect on the microcontroller.

Example:

```
let dirs = %00000011 ` switch pins 0 and 1 to outputs
let pins = %00000011 ` switch on outputs 0 and 1
```

let: pins: / pinsc: =

Syntax:

{LET} pins = value

{LET} pinsc = value

- Value(s) are variables/constants which operate on the output port.

Function:

Set/clear all outputs on the main output port (let pins =).

Set/clear all outputs on portc (let pinsc =)

Information:

High and low commands can be used to switch individual outputs high and low. However when working with multiple outputs it is often convenient to change all outputs simultaneously. When working with this statement it is conventional to use binary notation. With binary notation output7 is on the left and output0 is on the right. If the bit is set to 0 the output will be off (low), if the bit is set to 1 the output will be on (high).

Note that this command will only function on pins configured as outputs. In this case it is necessary to configure the pins as outputs (using a let dirs = command) before use of this command.

Example:

```
let pins = %10000011 ` switch outputs 7,0,1 on
pause 1000 ` wait 1 second
let pins = %00000000 ` switch all outputs off
```

lookdown:

Syntax:

LOOKDOWN target,(value0,value1...valueN),variable

- Target is a variable/constant which will be compared to Values.
- Values are variables/constants.
- Variable receives the result (if any).

Function:

Get target's match number (0-N) into variable (if match found).

Information:

The lookdown command should be used when you have a specific value to compare with a pre-known list of options. The target variable is compared to the values in the bracket. If it matches the 5th item (value4) the number '4' is returned in variable. Note the values are numbered from 0 upwards (not 1 upwards). If there is no match the value of variable is left unchanged. In this example the variable b2 will contain the value 3 if b1 contains "d" and the value 4 if b1 contains "e"

Example:

```
lookdown b1, ( ``abcde`` ), b2
```

lookup:

Syntax:

LOOKUP offset,(data0,data1...dataN),variable

- Offset is a variable/constant which specifies which data# (0-N) to place in Variable.
- Data are variables/constants.
- Variable receives the result (if any).

Function:

Lookup data specified by offset and store in variable (if in range).

Description:

The lookup command is used to load variable with different values. The value to be loaded in the position in the lookup table defined by offset. In this example if b0 = 0 then b1 will equal "a", if b0 =1 then b1 will equal "b" etc. If offset exceeds the number of entries in the lookup table the value of variable is unchanged.

Example:

```
main:
    lookup b0, ( ``abcd`` ), b1    ` put ASCII character into b1
    inc b0                        ` increment b0
    if b0 < 4 then main            ` loop
end
```

low:

Syntax:

LOW pin {,pin,pin...}

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.

Function:

Make pin output low.

Information:

The low command switches an output off (low). This command also automatically configures the pin as an output.

Example:

```
main:      high 1          'switch on output 1
           pause 5000      'wait 5 seconds
           low 1           'switch off output 1
           pause 5000      'wait 5 seconds
           goto main       'loop back to start
```

nap:

Syntax:

NAP period

- Period is a variable/constant which determines the duration of the reduced power nap (0-7).

Function:

Nap for a short period. Power consumption is reduced, but some timing accuracy is lost. A longer delay is possible with the sleep command.

Information:

The nap command puts the microcontroller into low power mode for a short period of time. When in low power mode all timers are switched off and so the pwmout and servo commands will cease to function (see the 'doze' command). The nominal period of time is given by this table. Due to tolerances in the microcontrollers internal timers, this time is subject to -50 to +100% tolerance. The external temperature affects these tolerances and so no design that requires an accurate time base should use this command.

Affect of increased clock speed:

The nap command uses the internal watchdog timer which is not affected by changes in resonator clock speed.

Example: Use Button Circuit

```
main:
  high 1 ` switch on output 1
  nap 4 ` nap for 288ms
  low 1 ` switch off output 1
  nap 7 ` nap for 2.3 s
  goto main ` loop back to start
```

Period Time Delay

0 = 18ms
1 = 32ms
2 = 72ms
3 = 144ms
4 = 288ms
5 = 576ms
6 = 1.152 s
7 = 2.304 s

on...goto:

Syntax:

ON offset GOTO address0,address1...addressN

- Offset is a variable/constant which specifies which Address# to use (0-N).

- Addresses are labels which specify where to go.

Function:

Branch to address specified by offset (if in range).

Information:

This command allows a jump to different program positions depending on the value of the variable 'offset'. If offset is value 0, the program flow will jump to address0, if offset is value 1 program flow will jump to address1 etc. If offset is larger than the number of addresses the whole command is ignored and the program continues at the next line. This command is identical in operation to branch.

Example: Use Button Circuit

```
reset1:
  let b1 = 0          'set initial conditions
```

```

        low 0                'Green LED off
        low 1                'Red LED off
        if pin3 = 0 then goto reset1 'loop if button pressed
main:
    inc b1                    'Add 1 to b1
    pause 1000                'wait 1 second
    if b1 > 3 then reset1      'If end of cycle start over
    on b1 goto btn0,btn1, btn2, btn3 'ON-GOTO FUNCTION
    goto main                  'continue process
btn0:                          'Condition 0
    high 0                     'Green LED on
    goto main
btn1:                          'Condition 1
    high 1                     'Red LED on
    low 0                      'Green LED off
    goto main
btn2:                          'Condition 2
    high 0                     'Green LED on
    low 1                      'Red LED off
    goto main
btn3:                          'Condition 3
    high 0                     'Green LED on
    high 1                     'Red LED on
    goto main

```

on...gosub:

Syntax:

ON offset GOSUB address0, address1, ...addressN

- Offset is a variable/constant which specifies which subprocedure to use (0-N).
- Addresses are labels which specify which subprocedure to gosub to.

Function:

gosub address specified by offset (if in range).

Information:

This command allows a conditional gosub depending on the value of the variable 'offset'. If offset is value 0, the program flow will gosub to address0, if offset is value 1 program flow will gosub to address1 etc. If offset is larger than the number of addresses the whole command is ignored and the program continues at the next line. The return command of the sub procedure will return to the line after on...gosub. This command counts as a single gosub within the compiler.

Example: Use Button Circuit

```

reset1:
    let b1 = 0                'set initial conditions
    low 0                     'Green LED off
    low 1                     'Red LED off
    if pin3 = 0 then goto reset1 'loop if button pressed
main:
    inc b1                    'Add 1 to b1
    pause 1000                'wait 1 second
    if b1 > 3 then reset1      'If end of cycle start over

```

```

        on b1 gosub btn0,btn1, btn2, btn3  'ON-GOSUB FUNCTION
        goto main                        'continue process

btn0:                                     'Condition 0
    high 0                               'Green LED on
    return

btn1:                                     'Condition 1
    high 1                               'Red LED on
    low 0                                'Green LED off
    return

btn2:                                     'Condition 2
    high 0                               'Green LED on
    low 1                                'Red LED off
    return

btn3:                                     'Condition 3
    high 0                               'Green LED on
    high 1                               'Red LED on
    return

```

output:

Syntax:

OUTPUT pin,pin, pin...

- Pin is a variable/constant which specifies the i/o pin to use.

Function:

Make pin an output.

Information:

This command is only required on microcontrollers with programmable input/output pins (such as the PICAXE-08M). This command can be used to change a pin that has been configured as an input to an output. All pins are configured as inputs on first power-up (unless the pin is a fixed output). Fixed pins are not affected by this command. These pins are:

pin0 = fixed output, pin 3 = fixed input

Example:

```

main:
    input 1          'make pin input
    reverse 1        'make pin output
    reverse 1        'make pin input
    output 1         'make pin input

```

pause:

Syntax:

PAUSE milliseconds

- Milliseconds is a variable/constant (0-65535) which specifies how many milliseconds to pause (at 4MHz).

Function:

Pause for some time. The duration of the pause is as accurate as the resonator time-base, and presumes a 4MHz resonator.

Information:

The pause command creates a time delay (in milliseconds). The longest time delay possible is

just over 65 seconds. To create a longer time delay (for example 5 minutes) use a for...next loop

```
for b1 = 1 to 5      '5 loops
pause 60000          'wait 60 seconds
next b1              'Do this 4 more times to get 5 minutes
```

During a pause the only way to react to inputs is via an interrupt (see the setint command for more information). Do not put long pauses within loops that are scanning for changing input conditions. When using time delays longer than 5 seconds it may be necessary to perform a 'hard reset' to download a new program to the microcontroller. See the Serial Download section for more details.

Affect of increased clock speed:

The timebase is reduced to 0.5ms at 8MHz and 0.25ms at 16MHz .

Example: Use Button Circuit

```
main:
  high 1          'switch on output 1
  pause 5000      'wait 5 seconds
  high 0          'switch on output 0
  pause 5000      'wait 5 seconds
  low 1, 0        'Turn 1 & 0 off
  goto main       'loop back to start
  pause 2000      'Wait 2 seconds
```

peek:

Syntax:

PEEK location,variable,variable,WORD wordvariable...

- Location is a variable/constant specifying a register address. Valid values are 0 to 255 (see below).
- Variable is a byte variable where the data is returned. To use a word variable the keyword WORD must be used before the wordvariable name)

Function:

Read data from the microcontroller registers. This allows use of additional storage variables not defined by the bxx variables.

Information:

The function of the poke/peek commands is two fold. The most commonly used function is to store temporary byte data in the microcontrollers spare 'storage variable' memory. This allows the general purpose variables (b0, b1 etc.) to be re-used in calculations. Addresses 80 (\$50 in Hexadecimal) to 126 (\$7E in Hexadecimal) are general purpose registers that can be used freely.

The second function of the peek command is for experienced users to study the internal microcontroller SFR (special function registers). Addresses 0 (\$00) to 31 (\$1F) and 128 (\$80) to 159 (\$9F) are special function registers (for example - PORTB) which determine how the microcontroller operates. Avoid using these addresses unless you know what you are doing!

Example:

```
peek 80,b1          'put value of register 80 into variable b1
peek 80, word w1     'put value of register 80, 81 into variable W1
```

play:

Syntax:

PLAY tune,LED

- Output is fixed to pin 2.
- Tune is a constant (0 - 3) which specifies which tune to play

0 - Happy Birthday

1 - Jingle Bells

2 - Silent Night

3 - Rudolph the Red Nosed Reindeer

- LED is a constant (0 -3) which specifies if other outputs flash at the same time as the tune is being played.

0 - No outputs

1 - Output 0 flashes on and off

2 - Output 4 flashes on and off

3 - Output 0 and 4 flash alternately

Function:

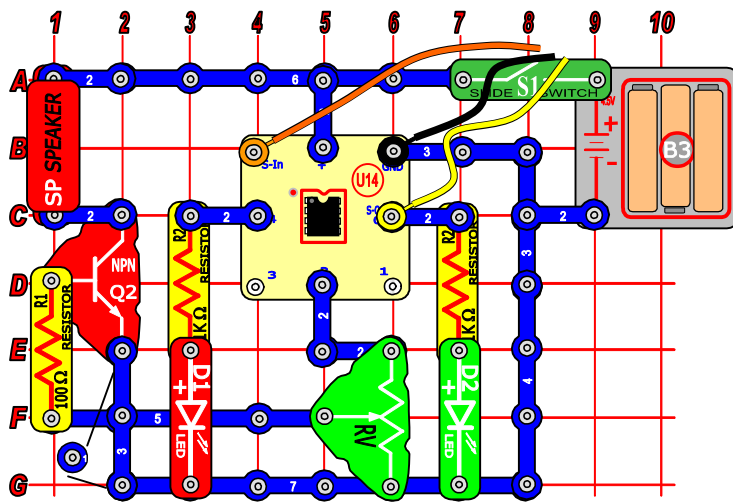
Play an internal tune out of the PICAXE output pin.

Description:

A microcontroller can play musical tones. The PICAXE08M2 is supplied with 4 pre-programmed internal tunes, which can be output via the play command. As these tunes are already included within the PICAXE bootstrap code, they use very little user program memory. To generate your own tunes use the 'tune' command, which can play any "mobile phone" style RTTTL tune.

Example: Use PLAY circuit below

```
main:      play b0,3      'play song with outputs 0 and 4 flashing
           inc b0         'get next song number      pause 2000      'wait
2 seconds goto main      'go back and play next song
```



When used with above program, this circuit will play the four songs imbedded in the 08M microcontroller and then repeat playing even when the variable b0 is greater than 3. The Play command will make the 4=0, 5=1, 6=2, 7=3, 8=0,,252=0, 253=1, 254=2, 255=3, 0=0 etc. Thus the songs will rotate until circuit is turned off.

poke:

Syntax:

POKE location,data,data,WORD wordvariable...

- Location is a variable/constant specifying a register address. Valid values are 0 to 255.
- Data is a variable/constant which provides the data byte to be written. To use a word variable the keyword WORD must be used before the wordvariable)

Function:

Write data into FSR location. This allows use of registers not defined by b0, b1 etc.

Information:

The function of the poke/peek commands is two fold.

The most commonly used function is to store temporary byte data in the microcontrollers spare 'storage variable' memory. This allows the general purpose variables (b0,b1 etc) to be re-used in calculations. Remember that to save a word variable two separate poke/peek commands will be required - one for each of the two bytes that form the word. Addresses 80 (\$50 in Hexadecimal) to 126 (\$7E in Hexadecimal) are general purpose registers that can be used freely. **The second function of the poke command is for experienced users to write the internal microcontroller SFR (special function registers).** Addresses 0 (\$00) to 31 (\$1F) and 128 (\$80) to 159 (\$9F) are special function registers (for example - PORTB) which determine how the microcontroller operates. **Avoid using these addresses unless you know what you are doing!**

Example:

```
poke 80,b1      'put value of variable b1 into register 80.
poke 80, word w1 'put value of variable W1 into register 80, 81
```

pulsin:

Syntax:

PULSIN pin, state, wordvariable

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.

- State is a variable/constant (0 or 1) which specifies which edge must occur before beginning the measurement in 10us units (4MHz resonator).
- Wordvariable receives the result (1-65535). If timeout occurs (0.65536s) the result will be 0.

Function:

Measure the length of an input pulse.

Information:

The pulsins command measures the length of a pulse. In no pulse occurs in the timeout period, the result will be 0. If state = 1 then a low to high transition starts the timing, if state = 0 a high to low transition starts the timing. Use the count command to count the number of pulses with a specified time period. It is normal to use a word variable with this command.

Affect of Increased Clock Speed:

4MHz 10us unit 0.65536s timeout

8MHz 5us unit 0.32768s timeout

16MHz 2.5us unit 0.16384s timeout

Example:

```
pulsin 3,1,w1 ` record the length of a pulse on pin 3 into w1
```

pulsout:

Syntax:

PULSOUT pin,time

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.
- Time is a variable/constant which specifies the period (0-65535) in 10us units (4MHz resonator).

Function:

Output a timed pulse by inverting a pin for some time.

Information:

The pulsout command generates a pulse of length time. If the output is initially low, the pulse will be high, and vice versa. This command automatically configures the pin as an output, but for reliable operation on 8 pin PICAXE you should ensure this pin is an output before using the command.

Affect of Increased Clock Speed:

4MHz 10us unit

8MHz 5us unit

16MHz 2.5us unit

Example:

main:

```
pulsout 1,150      'send a 1.50ms pulse out of pin 1
pause 20           'pause 20 ms
goto main          'loop back to start
```

random:

Syntax:

RANDOM wordvariable

- Wordvariable is both the workspace and the result. As random generates a pseudo-random sequence it is advised to repeatedly call it within a loop. A word variable **must** be used, byte variables will not operate correctly.

Function:

Generate next pseudo-random number in a word variable.

Description:

The random command generates a pseudo-random sequence of numbers between 0 and 65535. All microcontrollers must perform mathematics to generate random numbers, thus the sequence can never be truly random. On computers a changing quantity (such as the date/time) is used as the start of the calculation, so that each random command is different. The PICAXE does not contain such date functionality, and so the sequence it generates will always be identical unless the value of the word variable is set to a different value before the random command is used.

Another common way to overcome this issue (can be used on all parts) is to repeatedly call the random command within a loop, for example while waiting for a switch push. As the number of loops will vary between switch pushes, the output is much more random. If you only require a byte variable (0-255), still use the word variable (such as w0) in the command. As w0 is made up of b0 and b1, you can use either of these two bytes as your desired random byte variable.

Example:

```
main:                                ' note random is repeatedly called
random w0                            ' within the loop
if pin1 = 1 then doit
goto main
doit:
let pins = b1                        ' put random byte value on output pins
pause 100                            ' wait 0.1s
goto main                            ' loop back to start
```

readadc:

Syntax:

READADC channel,variable

- channel is a variable/constant specifying the ADC pin
- Variable receives the data byte read.

Function:

Read the ADC channel (8 bit resolution) contents into variable.

Information:

The readadc command is used to read the analogue value from the microcontroller input pins. Note that not all inputs have internal ADC functionality. On the 08M microcontroller with 'shared' inputs the ADC pin is also a digital input pin. There are three 10 bit shared pins namely 1,2, and 4. The resolution of ADC is also shown in the table. The readadc command is used to read all types. However, with 10 bit ADC types the reading will be rounded to a byte value (8 bits). Use the readadc10 command to read the full 10 bit value. Low-resolution ADC inputs are based upon the microcontrollers internal 16 step comparator rather than the conventional internal ADC module. The reference voltage is normally the supply voltage (4.5V).

An 8-bit resolution analogue input will provide 256 different analogue readings (0 to 255) over the full voltage range (0 to 4.5V). A low-resolution analogue input will provide 16 readings over the lower two-thirds of the voltage range (0 to 3.0V). No readings are available in the upper third of the voltage range.

Standard 8 Bit Reading Low Resolution (16 Readings)

- channel is a variable/constant specifying the input pin (1-4)
- wordvariable receives the data word read.

Function:

Read the ADC channel (10 bit resolution) contents into wordvariable. Configuration is automatic
Information:

The readadc10 command is used to read the analogue value from microcontrollers with 10-bit capability. Note that not all inputs have internal ADC functionality. The result is 10 bit, therefore a word variable must be used - for a byte value use the readadc command instead.

Example:

```
main:
readadc10 1,w1 'read value into b3, b2
debug w1      'transmit to computer
pause 200     'short delay
goto main     'loop back to start
```

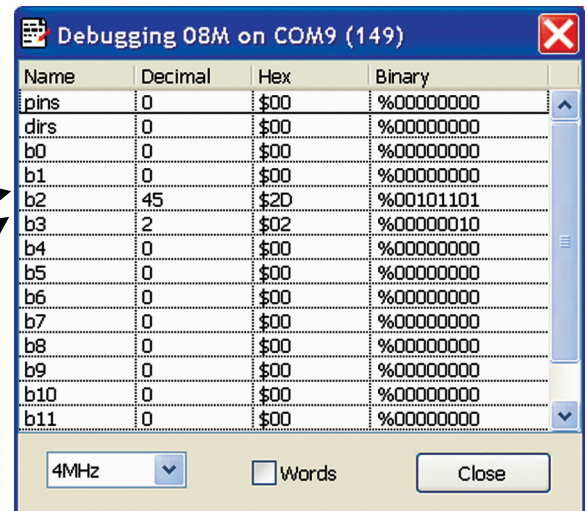
Least significant part of reading
Most significant part of reading (times 255)

Reading = (255x2) + 45 = 555

Minimum reading = 0

Maximum reading = (3x255) + 255 = 1020

Debug screen opens after download or press F6.



Name	Decimal	Hex	Binary
pins	0	\$00	%00000000
dirs	0	\$00	%00000000
b0	0	\$00	%00000000
b1	0	\$00	%00000000
b2	45	\$2D	%00101101
b3	2	\$02	%00000010
b4	0	\$00	%00000000
b5	0	\$00	%00000000
b6	0	\$00	%00000000
b7	0	\$00	%00000000
b8	0	\$00	%00000000
b9	0	\$00	%00000000
b10	0	\$00	%00000000
b11	0	\$00	%00000000

4MHz ☐ Words

read:

Syntax:

READ location,variable,variable, WORD wordvariable

- Location is a variable/constant specifying a byte-wise address (0-255).
- Variable receives the data byte read. To use a word variable the keyword WORD must be used before the wordvariable)

Function:

Read EEPROM data memory byte content into variable.

Information:

The read command allows byte data to be read from the microcontrollers data memory. The contents of this memory is not lost when the power is removed. However the data is updated (with the EEPROM command specified data) upon a new download. To save the data during a program use the write command. The read command is byte wide, so to read a word variable two separate byte read commands will be required, one for each of the two bytes that makes the word (for example to variable w0, read both b0 and b1). With the PICAXE 08M2, the data memory is shared with program memory. Therefore only unused bytes may be used within a program. To establish the length of the program use 'Check Syntax' from the PICAXE menu. This will report the length of program. Available data addresses can then be used as follows:

PICAXE-08M2 (255 - number of used bytes). When word variables are used (with the keyword WORD) the two bytes of the word are saved/retrieved in a little endian manner (ie low byte at address, high byte at address + 1)

Example:

```

main:
for b0 = 0 to 63      'start a loop
read b0,b1           'read value at b0 into b1
serout 7,T2400,(b1)  'transmit value to serial LCD
next b0              'next loop

```

readoutputs

Syntax:

READOUTPUTS variable

- variable is a byte variable to receive the output pins values

Function:

Read the output pins value into variable.

Information:

The current state of the output pins can be read into a variable using the readoutputs command. Note that this is not the same as 'let var = pins', as this let command reads the status of the input (not output) pins.

Example:

```

main:
readoutputs b1 'read outputs value into b1

```

return:

Syntax:

RETURN

Function:

Return from subroutine.

Information:

The return command is only used with a matching 'gosub' command, to return program flow back to the main program at the end of the sub procedure. If a return command is used without a matching 'gosub' beforehand, the program flow will crash.

Example:

```

main:
let b2 = 15          'set b2 value
pause 2000           'wait for 2 seconds
gosub flsh           'call sub-procedure
let b2 = 5           'set b2 value
pause 2000           'wait for 2 seconds
gosub flsh           'call sub-procedure
end                  'stop accidentally falling into sub
flsh:
for b0 = 1 to b2      'define loop for b2 times
high 1               'switch on output 1
pause 500            'wait 0.5 seconds
low 1                'switch off output 1
pause 500            'wait 0.5 seconds
next b0              'end of loop

```

return **'return from sub-procedure**

reverse:

Syntax:

REVERSE pin, pin, pin...

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.

Function:

Make pin an output if now input and vice versa.

Information:

This command is only required on microcontrollers with programmable input/output pins like the PICAXE-08M2. This command can be used to change a pin that has been configured as an input to an output. All pins are configured as inputs on first power-up (unless the pin is a fixed output). Fixed pins are not affected by this command. These pins are pin 0 = fixed output, and pin 3 = fixed input.

Example:

main:

```
input 1                    'make pin input
reverse 1                  'make pin output
reverse 1                  'make pin input
output 1                   'make pin output
```

select: case: \ case: \ else: \ endselect:

Syntax:

```
SELECT VAR
CASE VALUE
{code}
CASE VALUE, VALUE...
{code}
CASE VALUE TO VALUE
{code}
CASE ?? value
{code}
ELSE
{code}
ENDSELECT
```

- Var is the value to test.

- Value is a variable/constant.

?? can be any of the following conditions;

= equal to	> greater than
is equal to	>= greater than or equal to
<> not equal to	< less than
!= not equal to	<= less than or equal to

Function:

Compare a variable value and conditionally execute sections of code.

Information:

The multiple select \ case \ else \endselect command is used to test a variable for certain conditions. If these conditions are met that section of the program code is executed, and then program flow jumps to the endselect position. If the condition is not met program flows jumps directly to the next case or else command. The 'else' section of code is only executed if none of the case conditions have been true.

Example:

```
select case b1 'Test variable b1
case 1         'If b1 = 1
high 1         'Make pin 1 high
case 2,3       'If b1 = 2 or 3
low 1          'Make pin 1 low
case 4 to 6    'If b1 = 4 to 6
high 2         'Make pin 2 high
else          'all other values
low 2          'Make pin 2 low
endselect      'Test finished
```

serin: *For full description see "Picaxe Manual Part 2"*

Syntax:

SERIN pin,baudmode,(qualifier,qualifier...),{#}variable,{#}variable...

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.
- Baudmode is a variable/constant (0-7) which specifies the mode:
- Qualifiers are optional variables/constants (0-255) which must be received in exact order before subsequent bytes can be received and stored in variables.
- Variable(s) receive the result(s) (0-255). Optional #'s are for inputting ASCII decimal numbers into variables, rather than raw characters.

Function:

Serial input with optional qualifiers (8 data, no parity, 1 stop).

Information:

The serin command is used to receive serial data into an input pin of the microcontroller. It cannot be used with the serial download input pin, which requires use of the serrxd command instead. Pin specifies the input pin to be used. Baud mode specifies the baud rate and polarity of the signal. Qualifiers are used to specify a 'marker' byte or sequence. The command **serin 1,N2400,('`ABC`'),b1** requires the string "ABC" before the next byte read is put into byte b1. For **serin 1,N2400,b1** the first byte received will be put into b1 regardless.

All processing stops until the new serial data byte is received. This command cannot be interrupted by the setint command. The following example simply waits until the sequence "go" is received.

```
serin 1,N2400,('`go`')
```

Example Computer Interface Circuit:

PICAXE-08M2- Due to the internal structure of input 3 on this chip, a 1N4148 diode is required between the pin and V+ for serin to work on this particular pin ('bar' end of diode to V+) with

this circuit. All other pins have an internal diode.

Example:

```
main:for b0 = 0 to 63          'start a loop
serin 6,N2400,b1              'receive serial value
write b0,b1                   'write value into b1
next b0                       'next loop
```

serout: *For full description see "Picaxe Manual Part 2"*

Syntax:

SEROUT pin,baudmode,({#}data,{#}data...)

- Pin is a variable/constant (1,2,&4) which specifies the i/o pin to use.

- Baudmode is a variable/constant (0-7) which specifies the mode:

Txxx give a true output (idle high)

Nxxx give an inverted output (idle low)

- Data are variables/constants (0-255) which provide the data to be output.

Optional #'s are for outputting ASCII decimal numbers, rather than raw characters. Text can be enclosed in speech marks ("Hello")

Function:

Transmit serial data output (8 data bits, no parity, 1 stop bit).

Information:

The serout command is used to transmit serial data from an output pin of the microcontroller. It cannot be used with the serial download output pin - use the srtxd command in this case. Pin specifies the output pin to be used. Baud mode specifies the baud rate and polarity of the signal. When using simple resistor interface, use N (inverted) signals. When using a MAX232 type interface use T (true) signals. The protocol is fixed at N,8,1 (no parity, 8 data bits, 1 stop bit).

A 'N' baud rate idles low, with data pulse going high.

A 'T' baud rate idles high, with data pulses going low.

When using a T baud rate the very first byte may become corrupt if the output pin was low before the serout command (the pin will be automatically left high after the serout command). To avoid this issue place the line high (via a 'high' command) a few milliseconds before the very first serout command. The # symbol allows ASCII output. Therefore #b1, when b1 contains the data 126, will output the ascii characters "1" "2" "6" rather than the raw data 126.

Example:

```
main:
for b0 = 0 to 63          'start a loop
read b0,b1                'read value into b1
serout 7,N2400,(b1)       'transmit value to serial LCD
next b0                   'next loop
```

srtxd:

Syntax:

SERTXD ({#}data,{#}data...)

- Data are variables/constants (0-255) which provide the data to be output.

Function:

Serial output via the serout programming pin (baud 4800, 8 data, no parity, 1 stop).

Information:

The sertxd command is similar to the serout command, but acts via the serial output pin rather than a general output pin. This allows data to be sent back to the computer via the programming cable. This can be useful whilst debugging data - view the uploaded data in the PICAXE>Terminal window. There is an option within View>Options>Options to automatically open the Terminal windows after a download. The baud rate is fixed at 4800,n,8,1.

Example:

```
main:
for b1 = 0 to 63 ` start a loop
sertxd(`The value of b1 is `,#b1,13,10)
pause 1000
next b1 ` next loop
```

servo: servopos: For Advanced Users
see PICAXE Manual Part 2 – Basic Commands

setint:

Syntax:

SETINT OFF

SETINT input,mask

- input is a variable/constant (0-255) which specifies input condition.
- mask is variable/constant (0-255) which specifies the mask

Function:

Interrupt on a certain inputs condition.

Information:

The setint command causes a polled interrupt on a certain input pin condition. A polled interrupt is a quicker way of reacting to a particular input combination. It is the only type of interrupt available in the PICAXE system. The inputs port is checked between execution of each command line in the program, between each note of a tune command, and continuously during any pause command. If the particular inputs condition is true, a 'gosub' to the interrupt sub-procedure is executed immediately. When the sub-procedure has been carried out, program execution continues from the main program. The interrupt inputs condition is any pattern of '0's and '1's on the input port, masked by the byte 'mask'. Therefore any bits masked by a '0' in byte mask will be ignored.

To interrupt on input1 high only;

```
setint %00000010,%00000010
```

To interrupt on input1 low only;

```
setint %00000000,%00000010
```

To interrupt on input0 high, input1 high and input 2 low;

```
setint %00000011,%00000111
```

etc.

Only one input pattern is allowed at any time. To disable the interrupt execute a SETINT OFF command.

Notes:

- 1) Every program which uses the SETINT command must have a corresponding interrupt: sub-procedure (terminated with a return command) at the bottom of the program.
- 2) When the interrupt occurs, the interrupt is permanently disabled. Therefore to re-enable the interrupt (if desired) a SETINT command must be used within the interrupt: sub-procedure itself. The interrupt will not be enabled until the 'return' command is executed.
- 3) If the interrupt is re-enabled and the interrupt condition is not cleared within the sub-procedure, a second interrupt may occur immediately upon the return command.
- 4) After the interrupt code has executed, program execution continues at the next program line in the main program. In the case of the interrupted pause, wait, play or tune command, any remaining time delay is ignored and the program continues with the next program line.

More detailed SETINT explanation.

The SETINT must be followed by two numbers - a 'compare with value' (input) and an 'input mask' (mask) in that order. It is normal to display these numbers in binary format, as it makes it more clear which pins are 'active'. In binary format input7 is on the left and input0 is on the right. The second number, the 'input mask', defines which pins are to be checked to see if an interrupt is to be generated ...

- %00000001 will check input pin 0
- %00000010 will check input pin 1
- %00000100 will check input pin 2
- %00001000 will check input pin 3
- %00010000 will check input pin 4

These can also be combined to check a number of input pins at the same time...

- %00000011 will check input pins 1 and 0
- %00010100 will check input pins 4 and 2

Having decided which pins you want to use for the interrupt, the first number (inputs value) states whether you want the interrupt to occur when those particular inputs are on (1) or off (0). Once a SETINT is active, the PICAXE monitors the pins you have specified in 'input mask' where a '1' is present, ignoring the other pins. An input mask of %00010100 will check pins 4 and 2 and create a value of %000a0b00 where bit 'a' will be 1 if pin 7 is high and 0 if low, and bit 'b' will be 1 if pin 2 is high and 0 if low. The 'compare with value', the first argument of the SETINT command, is what this created value is compared with, and if the two match, then the interrupt will occur, if they don't match then the interrupt won't occur. If the 'input mask' is %00010100, pins 4 and 2, then the valid 'compare with value' can be one of the following ...

- %00000000 Pin 4 = 0 and pin 2 = 0
- %00000100 Pin 4 = 0 and pin 2 = 1
- %00010000 Pin 4 = 1 and pin 2 = 0
- %00010100 Pin 4 = 1 and pin 2 = 1

So, if you want to generate an interrupt whenever Pin 4 is high and Pin 2 is low, the 'input mask' is %00010100 and the 'compare with value' is %00010000, giving a SETINT command of ...

- SETINT %00010000,%00010100

The interrupt will then occur when, and only when, pin 4 is high and pin 2 is low. If pin 4 is low or pin 2 is high the interrupt will not happen as two pins are 'looked at' in the mask.

Example:

```
setint %00010000,%00010100  
` activate interrupt when pin4 only goes high  
main:
```

```

low 1 ` switch output 1 off
pause 2000 ` wait 2 seconds
goto main ` loop back to start
interrupt:
high 1 ` switch output 1 on
if pin7 = 1 then interrupt ` loop here until the
` interrupt cleared
pause 2000 ` wait 2 seconds
setint %10000000,%10000000 ` re-activate interrupt
return ` return from sub

```

In this example an LED on output 1 will light immediately the input is switched high. With a standard if pin7 =1 then.... type statement the program could take up to two seconds to light the LED as the if statement is not processed during the pause 2000 delay time in the main program loop (standard program shown below for comparison).

```

main:
low 1 ` switch output 1 off
pause 2000 ` wait 2 seconds
if pin7 = 1 then sw_on
goto main ` loop back to start
sw_on:
high 1 ` switch output 1 on
if pin7 = 1 then sw_on
` loop here until the condition is cleared
pause 2000 ` wait 2 seconds
goto main ` back to main loop

```

setfreq:

Syntax:

setfreq freq

- freq is the keyword that selects the appropriate internal frequency of 4 MHz or 8 MHz.

Function:

Set the internal clock frequency to 8MHz (m8) or 4MHz (m4) value.

The default value is 4MHz.

Information:

The setfreq command can be used to change the speed of operation of the microcontroller from 4MHz to 8MHz (or 8MHz to 4MHz). However note that this speed increase affects many commands, by, for instance, changing their properties (e.g. all pause commands are half the length at 8MHz). The change occurs immediately. All programs default to m4 (4MHz) if a setfreq command is not used. The internal resonator frequencies are factory preset to the most accurate settings. However advanced users may use the calibfreq command to adjust these factory preset settings. Some commands such as readtemp will only work at 4MHz. In these cases change back to 4MHz temporarily to operate these commands.

Note that a temporary change in frequency will have a direct affect on background frequency dependant tasks such as pwmout / hpwm.

Example:

```
setfreq m4 ` setfreq to 4MHz
```

```
readtemp 1,b1 ` do command at 4MHz
setfreq m8 ` set freq back to 8MHz
```

sleep:

Syntax:

SLEEP period

- Period is a variable/constant which specifies the duration of sleep in multiples of 2.3 seconds (1-65535).

Function:

Sleep for some period (multiples of approximately 2.3s).

Information:

The sleep command puts the microcontroller into low power mode for a period of time. When in low power mode all timers are switched off and so the pwmout and servo commands will cease to function. The nominal period is 2.3s, so sleep 10 will be approximately 23 seconds. The sleep command is not regulated and so due to tolerances in the microcontrollers internal timers, this time is subject to - 50 to +100% tolerance. The external temperature affects these tolerances and so no design that requires an accurate time base should use this command. Shorter 'sleeps' are possible with the 'nap' command (where supported). Some PICAXE chips support the disablebod (enablebod) command to disable the brown-out detect function. Use of this command prior to a sleep will considerably reduce the current drawn during the sleep command. The command 'sleep 0' is ignored.

Affect of increased clock speed:

The sleep command uses the internal watchdog timer which is not affected by changes in resonator clock speed.

Example:

```
main: high 1 ` switch on output 1
sleep 10 ` sleep for 23 seconds
low 1 ` switch off output 1
sleep 100 ` sleep for 230 seconds
goto main ` loop back to start
```

sound:

Syntax:

SOUND pin,(note,duration,note,duration...)

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.

- Note(s) are variables/constants (0-255) which specify type and frequency. Note 0 is silent for the duration. Notes 1-127 are ascending tones. Notes 128-255 are ascending white noises.

- Duration(s) are variables/constants (0-255) which specify duration (multiples of approx 10ms).

Function:

Play sound 'beep' noises.

Information:

This command is designed to make audible 'beeps' for games and keypads etc. To play music use the play or tune command instead. Note and duration must be used in 'pairs' within the command.

Affect of Increased Clock Speed:

The length of the note is halved at 8MHz.

Example:

```
main: let b0 = b0 + 1 ` increment b0
sound 7, (b0,50) ` make a sound
goto main ` loop back to start
```

stop:

Syntax:

STOP

Function:

Enter a permanent stop loop until the power cycles (program re-runs) or the PC connects for a new download.

Information:

The stop command places the microcontroller into a permanent loop at the end of a program. Unlike the end command the stop command does not put the microcontroller into low power mode after a program has finished. The stop command does not switch off internal timers, and so commands such as servo and pwmout that require these timers will continue to function.

Example:

```
main:
pwmout 1,120,400
stop
```

switch: on/off:

Syntax:

SWITCH ON pin, pin, pin...

SWITCHON pin, pin, pin...

SWITCH OFF pin, pin, pin...

SWITCHOFF pin, pin, pin...

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.

Function:

Make pin output high / low.

Information:

This is a 'pseudo' command designed for use by younger students. It is actually equivalent to 'high' or 'low', ie the software outputs a high or low command as appropriate.

Example:

```
main: switch on 7 ` switch on output 7
wait 5 ` wait 5 seconds
switch off 7 ` switch off output 7
wait 5 ` wait 5 seconds
goto main ` loop back to start
```

symbol:

Syntax:

SYMBOL symbolname = value

SYMBOL symbolname = value ?? constant

- Symbolname is a text string which must begin with an alpha-character or '_'. After the first

character, it can also contains number characters ('0'-'9').

- Value is a variable or constant which is being given an alternate symbolname.

- ?? can be any supported mathematical function e.g. + - * / etc.

Function:

Assign a value to a new symbol name.

Mathematical operators can also be used on constants (not variables)

Information:

Symbols are used to rename constants or variables to make them easier to remember during a program. Symbols have no affect on program length as they are converted back into 'numbers' before the download. Symbols can contain numeric characters, but must not start with a numeric character. Naturally symbol names cannot be command names or reserved words such as input, step, etc. See the list of reserved words at the end of this section. When using input and output pin definitions take care to use the term 'pin0' not '0' when describing input variables to be used within if...then statements.

Example:

```
symbol RED_LED = 7 ` define a output pin
symbol PUSH_SW = pin1 ` define a input switch
symbol DELAY = B0 ` define a variable symbol
let DELAY = 200 ` preload counter with 200
main: high RED_LED ` switch on output 7
pause DELAY ` wait 0.2 seconds
low RED_LED ` switch off output 7
pause DELAY ` wait 0.2 seconds
goto main ` loop back to start
```

toggle:

Syntax:

TOGGLE pin,pin,pin...

- Pin is a variable/constant (0-4) which specifies the i/o pin to use.

Function:

Make pin output and toggle state.

Information:

The high command inverts an output (high if currently low and vice versa) On microcontrollers with configurable input/output pins (e.g. PICAXE-08) this command also automatically configures the pin as an output.

Example:

```
main:
toggle 7 ` toggle output 7
pause 1000 ` wait 1 second
goto main ` loop back to start
```

tune:

Syntax:

TUNE LED, speed, (note, note, note...)

The i/o pin for tunes is fixed to output 2.

LED is a variable/constant (0 -3) which specifies if other outputs flash at the same time as the tune is being played.

0 - No outputs

- 1 - Output 0 flashes on and off
- 2 - Output 4 flashes on and off
- 3 - Output 0 and 4 flash alternately
- speed is a variable/constant (1-15) which specifies the tempo of the tune.
- notes are the actual tune data generated by the Tune Wizard.

Function:

Plays a user defined musical tune .

Information:

The tune command allows musical 'tunes' to be played. Playing music on a micro-controller with limited memory will never have the quality of commercial playback devices, but the tune command performs remarkably well. Music can be played on speaker using a transistor amplifier or on headsets. Technical details of the note encoding process can be found at the picaxe website. We will use the 'Tune Wizard' to automatically generate the tune command, by either manually sequentially entering notes or by importing a mobile phone ring tone. The longer the tune the more memory that will be used. The 'play' command does not use up memory in the same way, but is limited to the 4 internal preset tunes.

Speed:

The speed of music is normally called 'tempo' and is the number of 'quarter beats per minute' (BPM). This is defined within the PICAXE system by allocating a value of 1-15 to the speed setting. A table of different speed values are shown here. This gives a good range for most popular tunes.

Speed BPM

1 812
 2 406
 3 270
 4 203
 5 162
 6 135
 7 116
 8 101
 9 90
 10 81
 11 73
 12 67
 13 62
 14 58
 15 54

.

PICAXE Tune Wizard

The Tune Wizard allows musical tunes to be created for the PICAXE. This is not available with this editor but only with the picaxe editor found on the revolution web site.

There are approximately ??? tunes on the Snap Circuit CD. To download a tune you must highlight the code, click the 'Copy' button to copy the tune command to the Windows clipboard.

The tune can then be pasted into your main program.

Ring Tone Tips & Tricks:

1. After downloading the tune, try adjusting the tempo by increasing or decreasing the speed value by 1 and listening to which 'speed' sounds best.
2. If your ringtone does not import, make sure the song title at the start of the line is less than 50 characters long and that all the text is saved on a single line.
3. Ringtones that contain the instruction 'd=16' after the description, or that contain many notes starting with 16 or 32 (the odd one or two doesn't matter) will not play correctly at normal speed on the PICAXE. However they may sound better if you double the PICAXE processor speed by using a 'setfreq m8' command before the tune command.
4. The PICAXE import filters 'round-down' dotted notes (notes ending with '.'). You may wish to change these notes into longer notes after importing.

Sound Circuits for use with the play or tune command.

The proper sound circuits for playing tunes on a speaker or on your headsets is shown in the manual.

Ring Tones Text Transfer Language (RTTTL) file format specification

```
<name> <sep> [<defaults>] <sep> <note-command>+
<name> := <char>+ ; max length 10 characters PICAXE accepts up to 50
<sep> := ":"
<defaults> :=
<def-note-duration> |<def-note-scale> |<def-beats>
<def-note-duration> := "d=" <duration>
<def-note-octave> := "o=" <octave>
<def-beats> := "b=" <beats-per-minute>
; If not specified, defaults are
; duration = 4 (quarter note)
; octave = 6
; beats-per-minute = 63 (decimal value) PICAXE defaults to 62
<note-command> :=
[<duration>] <note> [<octave>] [<special-duration>] <delimiter>
<duration> :=
"1" | ; Full 1/1 note
"2" | ; 1/2 note
"4" | ; 1/4 note
"8" | ; 1/8 note
"16" | ; 1/16 note Not used - PICAXE changes to 8
"32" | ; 1/32 note Not used - PICAXE changes to 8
<note> :=
"C" |
"C#" |
"D" |
"D#" |
"E" |
```

"F" |
 "F#" |
 "G" |
 "G#" |
 "A" |
 "A#" |
 "B" | ; "H" can also be used *PICAXE exports using B*
 "P" ; pause
 <octave> :=
 "5" | ; Note A is 440Hz
 "6" | ; Note A is 880Hz
 "7" | ; Note A is 1.76 kHz
 "8" ; Note A is 3.52 kHz *Not used - PICAXE uses octave 7*
 <special-duration> :=
 "." ; Dotted note *Not used - PICAXE rounds down*
 <delimiter> := " , "

wait:

Syntax:

WAIT seconds

- Seconds is a constant (1-65) which specifies how many seconds to pause.

Function:

Pause for some time in whole seconds.

Information:

This is a 'pseudo' command designed for use by younger students. It is actually equivalent to 'pause * 1000', ie the software outputs a pause command with a value 1000 greater than the wait value. Therefore this command cannot be used with variables. This command is not normally used outside the classroom.

Example:

```

main:
switch on 7 ` switch on output 7
wait 5 ` wait 5 seconds
switch off 7 ` switch off output 7
wait 5 ` wait 5 seconds
goto main ` loop back to start
  
```

write:

Syntax:

WRITE location,data ,data, WORD wordvariable...

- Location is a variable/constant specifying a byte-wise address (0-255).

- Data is a variable/constant which provides the data byte to be written. To use a word variable the keyword WORD must be used before the wordvariable.

Function:

Write byte data content into data memory.

Information:

The write command allows byte data to be written into the microcontrollers data memory. The contents of this memory is not lost when the power is removed. However the data is updated

(with the EEPROM command specified data) upon a new download. To read the data during a program use the read command. With the PICAXE-08, 08M, 14M, 18 and 18M the data memory is shared with program memory. Therefore only unused bytes may be used within a program. To establish the length of the program use 'Check Syntax' from the PICAXE menu. This will report the length of program. Available data addresses can then be used as follows:

PICAXE-08M has 0 to 255 - number of used bytes.

When word variables are used (with the keyword WORD) the two bytes of the word are saved/retrieved in a little endian manner (ie low byte at address, high byte at address + 1)

Example:

```
main:
for b0 = 0 to 63 ` start a loop
serin 6,T2400,b1 ` receive serial value
write b0,b1 ` write value into b0
next b0 ` next loop
```