

Contents

Basic Commands	4
Introduction.....	4
PICAXE Programming Editor Software	4
Labels	5
Comments	5
Constants.....	6
Symbols	6
Directives	7
Variables - General	9
Variables - System	10
Variables - Scratchpad	11
Variables - Storage	12
Variables - Special function	12
Variables - Mathematics	19
Variables - Unary Mathematics	22
backward	24
bcdtoascii	25
bintoascii	26
branch	27
button	28
calibadc (calibadc10)	30
calibfreq	31
count	32
debug	33
dec	34
disablebod	35
disconnect	36
do...loop	37
doze	38
eprom (data)	39
enablebod	40
end	41
exit	42
for...next	43
forward	44
get	45
gosub	46
goto	47
hi2cin	48
hi2cout	49
hi2csetup	50
hi2csetup - slave mode	50
hi2csetup - master mode	51
halt	53
hibernate	54
high	56
high portc	57
hpwm	58
hpwmduty	60
hserin	61
hserout	62
hsersetup	63
hspiin (hshin)	65
hspiout (hshout)	66
hspisetaup	67
i2cslave	69

if...then \ elseif...then \ else \ endif	71
if...then {goto}	73
if...and/or...then {goto}	73
if...then exit	74
if...and/or...then exit	74
if...then gosub	75
if...and/or...then gosub	75
inc	77
infrain	78
infrain2	80
infraout	81
input	86
irin	87
irout	88
kbin	89
keyin	90
keyled (kbled)	92
let	93
let dirs =	95
let dirsc =	95
let pins =	96
let pinsc =	96
lookdown	97
lookup	98
low	99
low portc	100
nap	101
on...goto	102
on...gosub	103
output	104
owin	105
owout	106
pause	107
pauseus	108
peek	109
play	110
poke	111
pulsin	112
pulsout	113
put	114
pwm	115
pwm duty	116
pwmout	117
random	119
readadc	120
readadc10	122
readi2c	123
read	124
readmem	125
readtable	126
readoutputs	127
readtemp	128
readtemp12	129
readowclk	130
resetowclk	131
readownsn	132
reconnect	134
reset	135
return	136
reverse	137
run	138

select case \ case \ else \ endselect	139
serin	140
serrxd	143
serout	144
sertxd	146
servo	147
servopos	148
setint	149
setintflags	152
setfreq	153
settimer	154
shiftin (spiin)	156
shiftout (spiout)	159
sleep	161
sound	162
stop	163
swap	164
switch on/off	165
symbol	166
table	167
toggle	168
tune	169
wait	176
write	177
writemem	178
writei2c	179
Additional Reserved Keywords	180
Manufacturer Website:	181
Trademark:	181
Acknowledgements:	181

BASIC COMMANDS

Introduction.

The PICAXE manual is divided into three sections:

- Section 1 - Getting Started
- Section 2 - BASIC Commands
- Section 3 - Microcontroller interfacing circuits

This second section provides the syntax (with detailed examples) for all the BASIC commands supported by the PICAXE system. It is intended as a lookup reference guide for each BASIC command supported by the PICAXE system. As some commands only apply to certain size PICAXE chips, a diagram beside each command indicates the sizes of PICAXE that the command applies to.

When using the flowchart method of programming, only a small sub-set of the available commands are supported by the on-screen simulation. These commands are indicated by the corresponding flowchart icon by the description.

For more general information about how to use the PICAXE system, please see section 1 'Getting Started'.

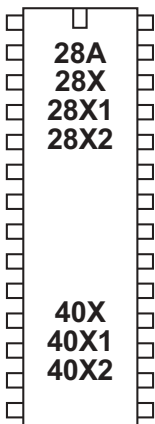
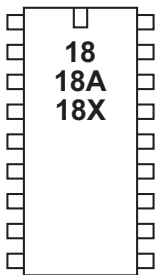
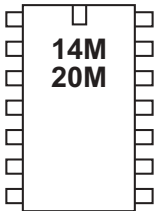
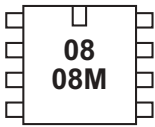
PICAXE Programming Editor Software

The software used for programming the PICAXE chips is called the 'Programming Editor'. This software is free to download from www.picaxe.co.uk. Please see section 1 of the manual ('Getting Started') for installation details and tutorials.

This manual was prepared using the '**enhanced compiler**' (View>Options>Editor menu) in version **5.2.0** of the Programming Editor software.

The latest version of the software is available on the PICAXE website 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.



Labels

Labels are used as markers throughout the program. Labels are used to mark a position in the program to 'jump to' at a later point using a goto, gosub or other command. Labels can be any word (that is not already a reserved keyword) and may contain digits and the underscore character. Labels must start with a letter (not digit), and are defined with a colon (:) at the marker position. The colon is not required within the actual commands.

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

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
```

Whitespace

Whitespace is the term used by programmers to define the white area on a printout of the program. 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.

Comments

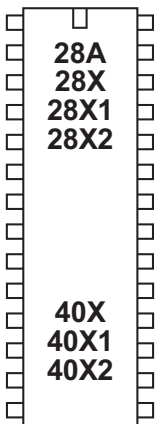
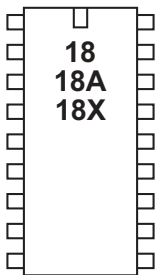
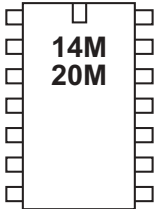
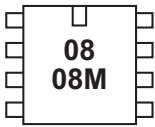
Comments are used to add information into the program for future reference. They are completely ignored by the computer during a download. Comments begin with an apostrophe (') or semi-colon (;) and continue until the end of the line. The keyword REM may also be used for a comment.

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

Examples:

```
high 0           \ make output 0 high
pause 1000      ; pause 1 second
low 0           REM make output 0 low

#rem
high 0
pause 1000
low 0
#endrem
```



Constants

Constants are 'fixed' numbers that are used within the program. The software supports word integers (any whole number between 0 and 65335).

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
%01100100   \ 01100100 binary
"A"          \ "A" ascii (65)
"Hello"      \ "Hello" - equivalent to "H","e","l","l","o"
B1 = B0 ^ $AA \ xor variable B0 with AA hex
```

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.

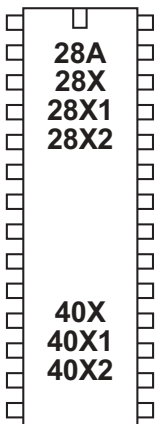
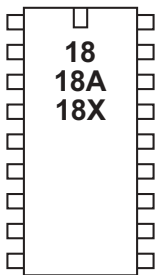
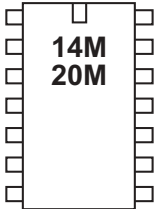
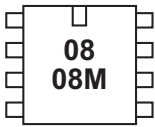
Symbols can use any word that is not a reserved keyword (e.g. switch, step, output, input, etc.)

Symbols can contain numeric characters and underscores (flash1, flash_2 etc.) but the first character cannot be a numeric (e.g. 1flash)

Use of symbol does not increase program length. See the symbol command entry later in this manual for more information.

Example:

```
symbol RED_LED = 7   \ define a constant symbol
symbol COUNTER = b0 \ define a variable symbol
let COUNTER = 200   \ preload variable with value 200
mainloop:           \ define a program address
                   \ address symbol end with colons
high RED_LED        \ switch on output 7
pause COUNTER        \ wait 0.2 seconds
low RED_LED          \ switch off output 7
pause COUNTER        \ wait 0.2 seconds
goto mainloop       \ loop back to start
```



Directives

Directives are used by the software to set the current PICAXE type and to determine which sections of the program listing are to be compiled. Directives are therefore not part of the PICAXE program, they are instructions to the software compiler.

All directives start with a # and must be used on a single line. Any other non-relevant line content after the directive is ignored.

Directives marked *Programming Editor Only* are only supported by the PICAXE Programming Editor software and will not work with third party applications.

#picaxe 08/08m/14m/20m/18/18a/18x/28/28a/28x/28x1/28x2/40x/40x1/40x2
Set the compiler mode. This directive also automatically defines a label of the PICAXE type e.g. #picaxe 08m is also the equivalent of #define 08m. If no #picaxe directive is used the system defaults to the currently selected PICAXE mode (View>Options>Mode menu within Programming Editor).

Example: `#picaxe 08m`

#com port_number (Windows)

#com "device_name" (Linux)

Set the serial COM port for downloading.

Example: `#com 2`

Example: `#com "/dev/ttyS1"`

Remember that on Linux systems the COM port device name is actually one less than the COM port, so COM1 is "/dev/ttyS0"

#freq m4/m8/m16

Set the system clock frequency for 28X/40X parts only.

Example: `#freq m8`

#define label

Defines a label to use in an ifdef or ifndef statements.

Example: `#define clock8`

Do not confuse the use of #define and symbol =

#define is a directive and, when used with #ifdef, determines which sections of code are going to be compiled.

'symbol = ' is a command used within actual programs to re-label variables and pins

#undefine label

Removes a label from the current defines list

Example: `#undefine clock8`

#ifdef / #ifndef label**#else****#endif**

Conditionally compile code depending on whether a label is defined (#ifdef) or not defined (#ifndef).

Example: **#define clock8**
 #ifdef clock8
 let b1 = 8
 #else
 let b1 = 4
 #endif

#error "comment"

Force a compiler error at the current position

Example: **#error "Code not finished!"**

#rem / #endrem

Comment out multiple lines of text. Must be lower case.

Example:
 #rem
 high 0
 pause 1000
 low 0
 #endrem

#include "filename"

Include code from a separately saved file within this program.

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

NOTE: Reserved for future use. Not currently implemented.

Programming Editor Only Directives**#gosubs 16/255** *Programming Editor Only*

Set the gosubs mode (16/255) on X parts.

Example: **#gosubs 16**

#sim axe101/axe102/axe103/axe105/axe107/axe092 *Programming Editor Only*

Use a 'real life project' on screen whilst simulating

Example: **#sim axe105**

#simspeed value *Programming Editor Only*

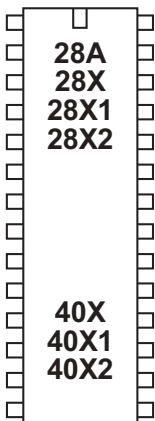
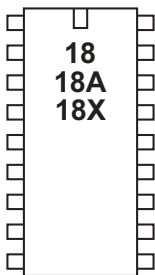
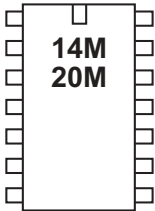
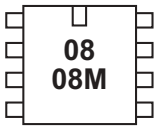
Set the simulation delay (in milliseconds)

Example: **#simspeed 200**

#terminal off/300/600/1200/4800/9600/19200/38400 *Programming Editor Only*

Config the Serial Terminal to open after a download (at selected baud rate)

Example: **#terminal 4800**



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
08	14	bit0-15	b0-13	w0-6
08M	14	bit0-15	b0-13	w0-6
14M	14	bit0-15	b0-13	w0-6
20M	14	bit0-15	b0-13	w0-6
18	14	bit0-15	b0-13	w0-6
18A	14	bit0-15	b0-13	w0-6
18X	14	bit0-15	b0-13	w0-6
28A	14	bit0-15	b0-13	w0-6
28X	14	bit0-15	b0-13	w0-6
28X1	28	bit0-31	b0-27	w0-13
28X2	48	bit0-31	b0-47	w0-23
40X	14	bit0-15	b0-13	w0-6
40X1	28	bit0-31	b0-27	w0-13
40X2	48	bit0-31	b0-47	w0-23

There are 14 (or more) 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 65535. 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 (32 on X1/X2) 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.

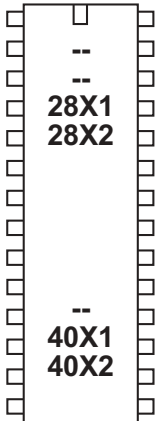
```
b0 = bit7: bit6: bit5: bit4: bit3: bit2: bit1: bit0
b1 = bit15: bit14: bit13: bit12: bit11: bit10: bit9: bit8
etc...
```

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.

All general purpose variables are reset to 0 upon a program reset.

Variables - System

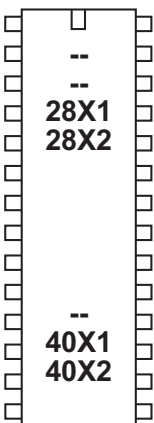
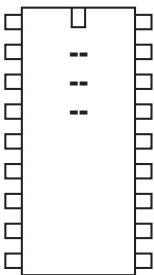
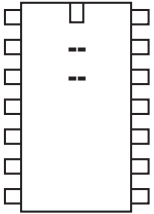
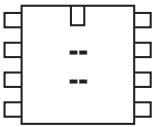
The X1 and X2 parts have 8 word variables, and 1 flags byte, which are reserved for system hardware use. However if that piece of system hardware is not used within a program the variables / flags may be used as general purpose variables.



s_w0	-	reserved for future use
s_w1	-	reserved for future use
s_w2	-	reserved for future use
s_w3	-	reserved for future use
s_w4	-	reserved for future use
s_w5	hserptr	hardware serial pointer
s_w6	hi2clast	hardware hi2c last byte written (slave mode)
s_w7	timer	timer value

The 'flags' byte variable is made up of 8 bit variables

flag0	-	reserved for future use
flag1	-	reserved for future use
flag2	-	reserved for future use
flag3	-	reserved for future use
flag4	-	reserved for future use
flag5	hserflag	hserial background receive has occurred
flag6	hi2cflag	hi2c write has occurred (slave mode)
flag7	toflag	timer overflow flag



Variables - Scratchpad

The scratchpad is a temporary memory area for storage of data such as arrays.

PICAXE-28X1, 40X1 parts have 128 scratchpad bytes (0-127)

PICAXE-28X2, 40X2 parts have 255 scratchpad bytes (0-255)

The scratchpad values can be addressed both directly and indirectly.

To directly address the scratchpad values the get (read the byte) and put (write the byte) commands are used.

To indirectly address the values the virtual variable name '@ptr' is used. @ptr is a variable name that can be used in any command (ie as where a 'b1' variable would be used). However the value of the variable is not fixed (as with b1) , but will contain the current value of the byte currently 'pointed to' by the scratchpad pointer (ptr).

Note that '@ptr' is a byte size variable, not a word size variable.

The compiler also accepts '@ptrinc' (post increment) and '@ptrdec' (post decrement) .

Every time the '@ptrinc' variable name is used in a command the value of the scratchpad pointer is automatically incremented by one (ie ptr = ptr+1 occurs automatically after the read/write of the value @ptr). This makes it ideal for storage of a single dimensional array of data.

```
ptr = 1           ` reset scratchpad pointer to 1
serrxd @ptrinc,@ptrinc,@ptrinc,@ptrinc,@ptr
                ` serin 5 bytes to scratchpad addresses 1-5

ptr = 1           ` reset scratchpad pointer to 1
for b1 = 1 to 5
  sertxd (@ptrinc) ` re-transmit those 5 values
next b1
```

See the put and get commands for more details.

Variables - Storage

Storage variables are additional memory locations allocated for temporary storage of byte data. They cannot be used in mathematical calculations, but can be used to temporarily store byte values by use of the peek and poke commands.

The number of available storage locations varies depending on PICAXE type. The following table gives the number of available byte variables with their addresses. These addresses vary according to technical specifications of the microcontroller. See the poke and peek command descriptions for more information.

PICAXE-08	none	
PICAXE-08M	48	80 to 127 (\$50 to \$7F)
PICAXE-14M,20M	48	80 to 127 (\$50 to \$7F)
PICAXE-18	48	80 to 127 (\$50 to \$7F)
PICAXE-18A	48	80 to 127 (\$50 to \$7F)
PICAXE-18X	96	80 to 127 (\$50 to \$7F), 192 to 239 (\$C0 to \$EF)
PICAXE-28A	48	80 to 127 (\$50 to \$7F)
PICAXE-28X, 40X	112	80 to 127 (\$50 to \$7F), 192 to 255 (\$C0 to \$FF)
PICAXE-28X1, 40X1	95	80 to 126 (\$50 to \$7E), 192 to 239 (\$C0 to \$EF)
PICAXE-28X2, 40X2	192	64 to 255 (\$40 to \$FF)

Variables - Special function

The special function variables available for use depend on the PICAXE type:

PICAXE-08 / 08M Special Function Registers

`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 08M `infrain2` command

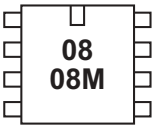
The variable `pins` is broken down into individual bit variables for reading from individual inputs with an `if...then` command. Only valid input pins are implemented.

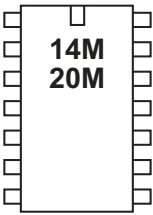
`pins` = `x : x : x : pin4 : pin3 : pin2 : pin1 : x`

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`





PICAXE-14M/20M Special Function Registers

pins = the input port when reading from the port
(out)pins = the output port when writing to the port
infra = a separate variable used within the infrain command
keyvalue = another name for infra, used within the keyin command

Note that pins is a 'pseudo' variable that can apply to both the input and output port.

When used on the left of an assignment pins applies to the 'output' port e.g.

```
let pins = %11000011
```

will switch outputs 7,6,1,0 high and the others low.

When used on the right of an assignment pins applies to the input port e.g.

```
let b1 = pins
```

will load b1 with the current state of the input port.

Additionally, note that

```
let pins = pins
```

means 'let the output port equal the input port'

To avoid this confusion it is recommended that the name 'outpins' is used in this type of statement e.g.

```
let outpins = pins
```

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.

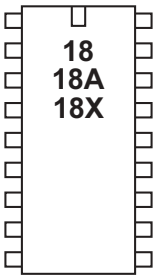
```

14M pins = x : x : x : pin4 : pin3 : pin2 : pin1 : pin0
20M pins = pin7 to pin0
  
```

The variable outpins is broken down into individual bit variables for writing outputs directly. Only valid output pins are implemented.

```

14M outpins = x : x : outpin5 : outpin4 : outpinx : outpin2 : outpin1 : outpin0
20M outpins = outpin7 to outpin0
  
```



PICAXE-18 / 18A / 18X Special Function Registers

pins = the input port when reading from the port
 (out)pins = the output port when writing to the port
 infra = a separate variable used within the infrain command
 keyvalue = another name for infra, used within the keyin command

Note that pins is a 'pseudo' variable that can apply to both the input and output port.

When used on the left of an assignment pins applies to the 'output' port e.g.

```
let pins = %11000011
```

will switch outputs 7,6,1,0 high and the others low.

When used on the right of an assignment pins applies to the input port e.g.

```
let b1 = pins
```

will load b1 with the current state of the input port.

Additionally, note that

```
let pins = pins
```

means 'let the output port equal the input port'

To avoid this confusion it is recommended that the name 'outpins' is used in this type of statement e.g.

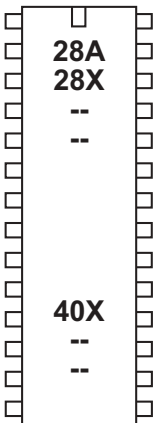
```
let outpins = pins
```

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 = pin7 : pin6 : x : x : x : pin2 : pin1 : pin0

The variable outpins is broken down into individual bit variables for writing outputs directly. Only valid output pins are implemented.

outpins = outpin7 : outpin6 : outpin5 : outpin4 :
outpin3 : outpin2 : outpin1 : outpin0



PICAXE-28A / 28X / 40X Special Function Registers

pins = the input port when reading from the port
 (out)pins = the output port when writing to the port
 infra = a separate variable used within the infrain command
 keyvalue = another name for infra, used within the keyin command

Note that pins is a 'pseudo' variable that can apply to both the input and output port.

When used on the left of an assignment pins applies to the 'output' port e.g.

```
let pins = %11000011
```

will switch outputs 7,6,1,0 high and the others low.

When used on the right of an assignment pins applies to the input port e.g.

```
let b1 = pins
```

will load b1 with the current state of the input port.

Additionally, note that

```
let pins = pins
```

means 'let the output port equal the input port'

To avoid this confusion it is recommended that the name 'outpins' is used in this type of statement e.g.

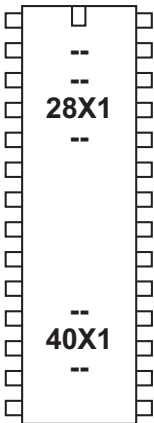
```
let outpins = pins
```

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 = pin7 : pin6 : pin5 : pin4 : pin3 : pin2 : pin1 : pin0

The variable outpins is broken down into individual bit variables for writing outputs directly. Only valid output pins are implemented.

outpins = outpin7 : outpin6 : outpin5 : outpin4 :
outpin3 : outpin2 : outpin1 : outpin0



PICAXE-28X1 / 40X1 Special Function Registers

pins = the input port when reading from the port
outpins = the output port when writing to the port
ptr = the scratchpad pointer
@ptr = the scratchpad value pointed to by ptr
@ptrinc = the scratchpad value pointed to by ptr (post increment)
@ptrdec = the scratchpad value pointed to by ptr (post decrement)
flags = system flags

When used on the left of an assignment 'outpins' applies to the 'output' port e.g.

```
let outpins = %11000011
```

will switch outputs 7,6,1,0 high and the others low.

When used on the right of an assignment 'pins' applies to the input port e.g.

```
let b1 = pins
```

will load b1 with the current state of the input port.

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 = pin7 : pin6 : pin5 : pin4 : pin3 : pin2 : pin1 : pin0
```

The variable outpins is broken down into individual bit variables for writing outputs directly. Only valid output pins are implemented.

```
outpins = outpin7 : outpin6 : outpin5 : outpin4 :  
outpin3 : outpin2 : outpin1 : outpin0
```

The scratchpad pointer variable is broken down into individual bit variables:

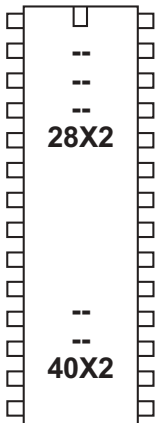
```
ptr = ptr7 : ptr6 : ptr5 : ptr4 : ptr3 : ptr2 : ptr1 : ptr0
```

See the 'Variables - Scratchpad' section for more information about

```
@ptr, @ptrinc, @ptrdec
```

The system 'flags' byte is broken down into individual bit variables. If the special hardware feature of the flag is not used in a program the individual flag may be freely used as a user defined bit flag.

Name	Special	Special function
flag0	-	reserved for future use
flag1	-	reserved for future use
flag2	-	reserved for future use
flag3	-	reserved for future use
flag4	-	reserved for future use
flag5	hserflag	hserial background receive has occurred
flag6	hi2cflag	hi2c write has occurred (slave mode)
flag7	toflag	timer overflow flag



PICAXE-28X2 / 40X2 Special Function Registers

pinsA	-the portA input pins
outpinsA	-the portA output pins
dirsA	- the portA data direction register
pinsB	- the portB input pins
outpinsB	-the portB output pins
dirsB	- the portB data direction register
pinsC	- the portC input pins
outpinsC	-the portC output pins
dirsC	- the portC data direction register
pinsD	- the portD input pins
outpinsD	-the portD output pins
dirsD	- the portD data direction register
ADC	- the ADC enable register (word)
ptr	- the scratchpad pointer
@ptr	- the scratchpad value pointed to by ptr
@ptrinc	- the scratchpad value pointed to by ptr (post increment)
@ptrdec	- the scratchpad value pointed to by ptr (post decrement)
flags	- system flags

When used on the left of an assignment 'outpins' applies to the 'output' pins e.g.

```
let outpinsB = %1100011
```

will switch outputs 7,6,1,0 high and the others low.

When used on the right of an assignment 'pins' applies to the input pins e.g.

```
let b1 = pinsB
```

will load b1 with the current state of the input pin on portB.

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

```
pinsB = pinB.7 : pinB.6 : pinB.5 : pinB.4 :
        pinB.3 : pinB.2 : pinB.1 : pinB.0
```

The variable outpinX is broken down into individual bit variables for writing outputs directly. Only valid output pins are implemented. e.g.

```
outpinsB = outpinB.7 : outpinB.6 : outpinB.5 : outpinB.4 :
            outpinB.3 : outpinB.2 : outpinB.1 : outpinB.0
```

The variable dirsX is broken down into individual bit variables for setting inputs/ outputs directly e.g.

```
dirsB = dirB.7 : dirB.6 : dirB.5 : dirB.4 :
        dirB.3 : dirB.2 : dirB.1 : dirB.0
```

The ADC enable register is a word value, not separated into separate bits. The whole word must always be read/written.

The scratchpad pointer variable is broken down into individual bit variables:

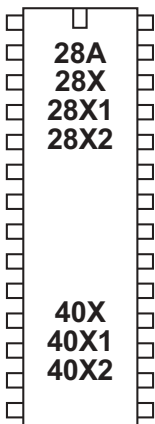
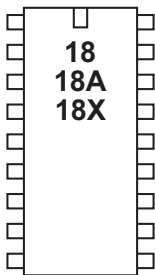
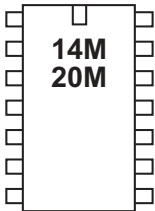
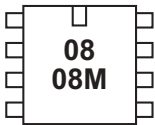
ptr = ptr7 : ptr6 : ptr5 : ptr4 : ptr3 : ptr2 : ptr1 : ptr0

See the 'Variables - Scratchpad' section for more information about

@ptr, @ptrinc, @ptrdec

The system 'flags' byte is broken down into individual bit variables. If the special hardware feature of the flag is not used in a program the individual flag may be freely used as a user defined bit flag.

Name	Special	Special function
flag0	-	<i>reserved for future use</i>
flag1	-	<i>reserved for future use</i>
flag2	-	<i>reserved for future use</i>
flag3	-	<i>reserved for future use</i>
flag4	-	<i>reserved for future use</i>
flag5	hserflag	hserial background receive has occurred
flag6	hi2cflag	hi2c write has occurred (slave mode)
flag7	toflag	timer overflow flag



Variables - Mathematics

The PICAXE microcontrollers support word (16 bit) mathematics. Valid integers are 0 to 65335. All mathematics can also be performed on byte (8 bit) variables (0-255), but if the result is greater than 255 overflow will occur without warning.

Maths is performed strictly from left to right. Unlike some computers and calculators, the PICAXE does not give * and / priority over + and -.

Therefore $3+4 \times 5$ is calculated as

$$3+4=7$$

$$7 \times 5=35$$

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.

$$\text{let } w1 = w2 / 5.7$$

is not valid, but

$$\text{let } w1 = w2 * 10 / 57$$

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)
//	%	; 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	
XNOR	^/	; bitwise XNOR	
ANDNOT	&/	; bitwise AND NOT	(NB this is <i>not</i> the same as NAND)
ORNOT	/	; bitwise OR NOT	(NB this is <i>not</i> the same as NOR)

The X1 and X2 parts also support

<<		; shift left	
>>		; shift right	
*/		; multiply	(returns middle word of result)
DIG		; return the digit value	
REV		; reverse a number of bits	

All mathematics is performed strictly from left to right.

On X1 and X2 parts it is possible to enclose part equations in brackets e.g.

```
let w1 = w2 / (b5 + 2)
```

On all other chips it is not possible to enclose part equations in brackets e.g.

```
let w1 = w2 / b5 + 2
```

is not valid. This would need to be entered in equivalent form e.g.

```
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, 0-65335 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 $Saabb \times Scdd = Seeffghh$

In PICAXE maths

```
Saabb * Scdd = Sgghh
```

```
Saabb ** Scdd = Seeff
```

The X1 and X2 parts also support return of the middle word

```
Saabb */ Scdd = Sffgg
```

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
```

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 ascii "0" to the digit value e.g.

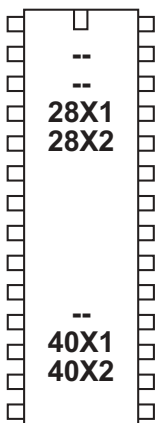
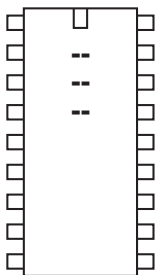
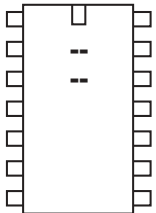
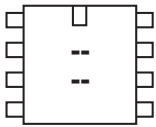
```
let b1 = b2 DIG 0 + "0"
```

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 = %00001101 REV 8
```



Variables - Unary Mathematics

The X1 and X2 parts also support these unary commands

SIN ; sine of angle (0 to 65535) in degrees
COS ; cosine of angle in degrees
SQR ; square root
INV ; invert / complement
NCD ; encoder (2n power encoder)
DCD ; decoder (2n power decoder)
BINTOBCD ; convert binary value to BCD
BCDTOBIN ; convert BCD value to binary

Further Information:

SIN and COS

The sin function returns a number equivalent to the sine of the value in degrees. The system uses a 45 step lookup table in each quadrant, giving a very fast, and reasonably accurate, result.

The sine function only works on positive whole integers. However as all sin and cos values repeat every 360 degrees, simply add 360 to make a negative value positive. e.g. $\sin(-30)$ is the same as $\sin(330)$ $(-30 + 360)$

As the real sine value is always a value between 1 and -1, a coding system is used to increase the accuracy when working with PICAXE whole integers. The value returned by the sin function is actually 100 x the real sine value. Therefore in normal mathematics $\sin 30 = 0.5$. In PICAXE mathematics this is returned as 50 $(100 * 0.5)$. This coding method provides a sine function accuracy equivalent to two decimal places.

e.g. `let b1 = sin 30` (answer b1 = 50)

Negative numbers are indicated by setting bit 7 of the returned byte. This has the effect of making negative values appear as 128 + the expected value.

e.g. `let b1 = sin 210` (answer b1 = 128+50 = 178)

The cos function operates in an identical manner.

SQR

The square root function returns the whole integer square root, according to 10 iterations of a N-R formula, using a seed of value/2. This formula provides a fast and accurate result. Note that as the PICAXE chip only operates with whole integers, the result will be rounded down to the nearest whole value.

e.g. `let b1 = sqr 64` (answer b1 = 8)

INV (~)

The invert function complements each bit in the value (ie each 0 is changed to a 1 and each 1 is changed to 0).

e.g. `let b1 = ~ %10101010` (answer b1 = %01010101)

NCD

The encoder function takes a value and finds the position of the highest bit in that number that is a 1. Therefore the result will be a number 1 to 16, when bit15 is 1 the answer is 1, when only bit0 is 1 the value is 16. If the value is 0, the result will be 0.

e.g `let b1 = ncd %00001000` (answer b1 = 13)

DCD

The decoder function takes a value between 0 and 15 and returns a 16 bit number, with that value bit set to 1.

e.g `let b1 = dcd 3` (answer b1 = %00001000)

BINTOBCD

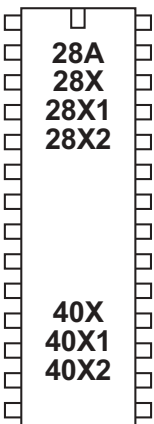
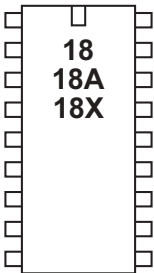
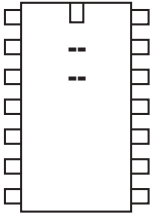
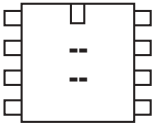
The bintobcd function converts a value to binary coded decimal. Note that the maximum value that can be returned within a byte is 99, or 9999 within a word.

e.g `let b1 = bintobcd 99` (answer b1 = %10011001=\$99)

BCDTOBIN

The bcdtobin function converts a binary coded decimal value to normal binary.

e.g `let b1 = bcdtobin $99` (answer b1 = 99)



backward

Syntax:

BACKWARD motor

- Motor is the motor name A or B.

Function:

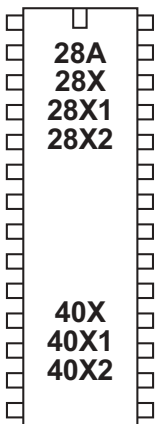
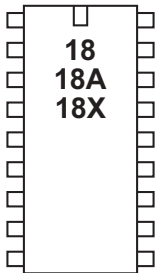
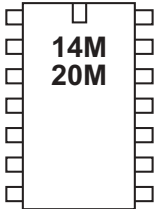
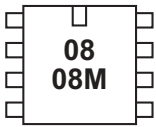
Make a motor output turn backwards

Information:

This is a 'pseudo' command designed for use by younger students with pre-assembled classroom models. It is actually equivalent to 'low 4 : high 5' (motor A) or 'low 6 : high 7' (motor B). This command is not normally used outside of the classroom.

Example:

```
main: forward A           \ motor a on forwards
      wait 5              \ wait 5 seconds
      backward A         \ motor a on backwards
      wait 5              \ wait 5 seconds
      halt A              \ motor A stop
      wait 5              \ wait 5 seconds
      goto main           \ loop back to start
```

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:

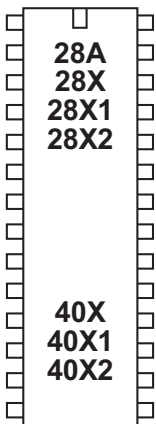
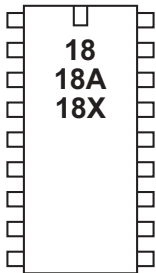
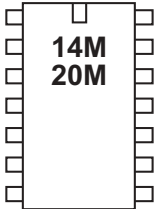
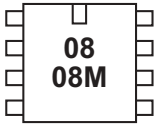
Convert a BCD value into separate ASCII bytes.

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:

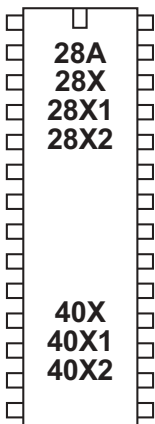
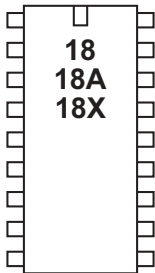
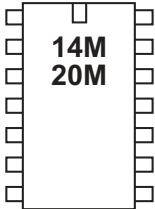
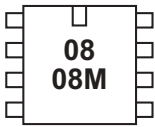
Convert a binary value into separate ASCII bytes.

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              ` loop back to start
```



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:

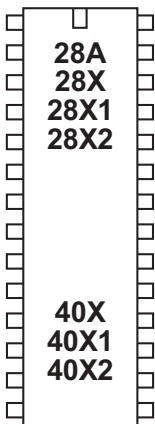
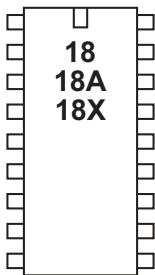
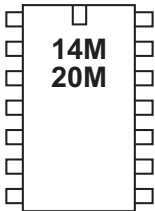
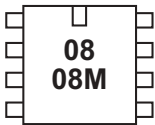
```

reset: let b1 = 0
       low 0
       low 1
       low 2
       low 3

main:  let b1 = b1 + 1
       if b1 > 3 then reset
       branch b1,(btn0,btn1, btn2, btn3)

btn0:  high 0
       goto main
btn1:  high 1
       goto main
btn2:  high 2
       goto main
btn3:  high 3
       goto main

```



button

Syntax:

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

- Pin is a variable/constant (0-7) 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 (0-255) which specifies time before a repeat if BUTTON is used within a loop.
- 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.
- 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 can cause microcontrollers to register multiple 'hits' with a single physical action, as the microcontroller can register each bounce as a new hit.

One simple way of overcoming this is to simply put a small pause (e.g. pause 10) within the program, this gives time for the switch to settle.

Alternately the button command can be used to overcome these issues. 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.

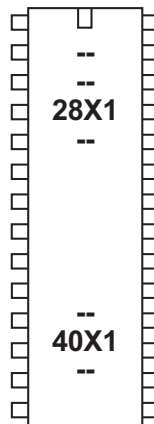
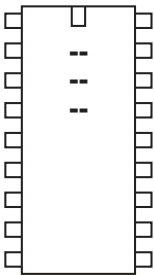
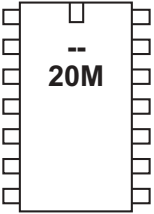
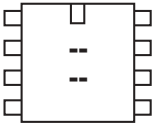
If the button command is within a loop, the next time the command is executed 'downstate' is once again checked. If the condition is still true, the variable 'bytevariable' is incremented. This can happen a number of times until 'bytevariable' value is equal to 'delay'. At this point a jump to 'address' 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', 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:

```
main:  button 0,0,200,100,b2,0,cont
                                     ` jump to cont unless pin0 = 0
        toggle 1                       ` else toggle input
        goto main
cont:  etc.
```



calibadc (calibadc10)

Syntax:

CALIBADC variable

CALIBADC10 wordvariable

- variable receives the adc reading

Function:

Calibrate the microcontrollers internal ADC by measuring a fixed (nominal) 0.6V source.

Information:

The reference voltage used by the PICAXE microcontrollers ADC reading (readadc/ readadc10) commands is the supply voltage. In the case of a battery powered system, this supply voltage can change over time (as the battery runs down), resulting in a varying ADC reading for the same voltage input.

The calibadc/calibadc10 commands can help overcome this issue by providing the ADC reading of a nominal internal 0.6V reference. Therefore by periodically using the calibadc command you can mathematically calibrate/compensate the readadc command for changes in supply voltage.

calibadc can be considered as 'carry out a readadc on a fixed 0.6V reference'

Note that the 0.6V is a nominal voltage only and will vary with each part.

Microchip datasheet AN1072 provides further details on how to software calibrate and use this advanced feature.

A formula to use this value is

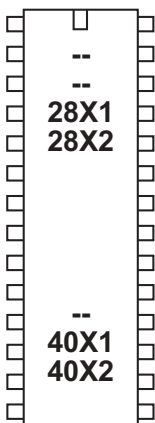
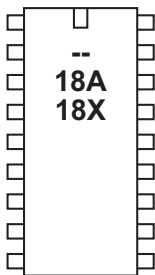
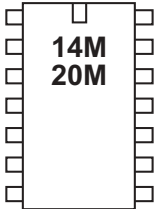
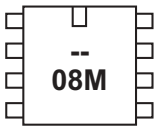
$$V_{\text{supply}} = \text{step} * 6 / \text{calib} / 10$$

where step = 255 (calib) or 1023 (calibadc10) and calib is the value returned from the calibadc command. Note that *6 / 10 is mathematically equivalent to multiply by 0.6 (the voltage reference).

Example:

main:

```
calibadc b1           \ read the 0.6V adc reading
debug b1             \ display current value
goto main            \ else loop back to start
```



calibfreq

Syntax:

CALIBFREQ {-} factor

- factor is a constant/variable containing the value -31 to 31

Function:

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

Information:

Some PICAXE chips have an internal resonator that can be set to 4 or 8Mhz operation via the setfreq command.

On these chips it is also possible to 'calibrate' this frequency. This is an advanced feature not normally required by most users, as all chips are factory calibrated to the most accurate setting. 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 value.

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.

The calibfreq is actually a pseudo command that performs a 'poke' command on the microcontrollers OSCTUNE register (address \$90).

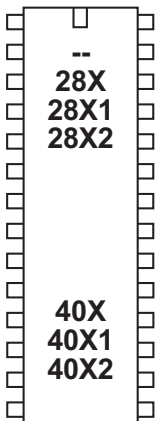
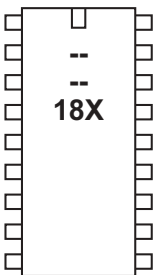
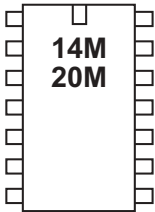
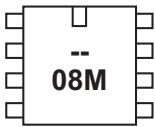
When the value is 0 to 31 the equivalent BASIC code is

```
poke $90, factor
pause 2
```

When the factor is -31 to -1 the equivalent BASIC code is

```
let b12 = 64 - factor
poke $90, b12
pause 2
```

Note that in this case variable b12 is used, and hence corrupted, by the command. This is necessary to poke the OSCTUNE register with the correct value.



count

Syntax:

COUNT pin, period, wordvariable

- Pin is a variable/constant (0-7) 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, so the highest frequency of pulses that can be counted is 25kHz, presuming a 50% duty cycle (ie equal on-off 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:

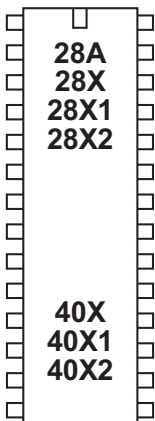
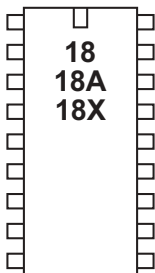
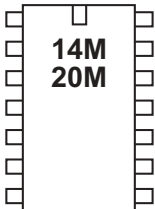
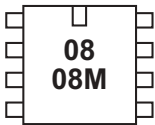
The period value is 0.5ms at 8MHz and 0.25ms at 16MHz.

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 (ie equal on-off 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 (ie equal on-off time).

Example:

```
main:
    count 1, 5000, w1      ` count pulses in 5 seconds
    debug w1              ` display value
    goto main             ` else loop back to start
```

debug



Syntax:

DEBUG {var}

- Var is an optional variable value (e.g. 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.

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 08/08M and 14M chips 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.

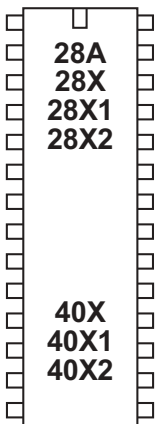
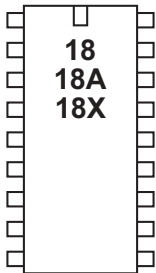
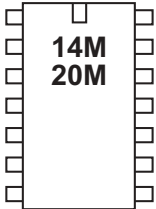
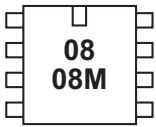
```
low 0           \ reset 0 to correct condition
pause 500      \ wait a while
debug b1       \ display values on computer screen
```

Affect of increased clock speed:

When using an 8 or 16Mhz clock speed ensure the software has been set with the correct speed setting to enable successful communication between microcontroller and PC.

Example:

```
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:

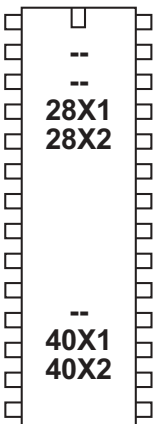
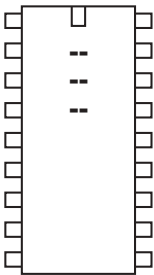
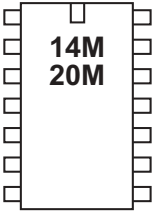
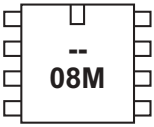
This command is shorthand for 'let var = var - 1'

Example:

```

for b1 = 1 to 5
  dec b2
next b1

```



disablebod

Syntax:

DISABLEBOD

Function:

Disable the on-chip brown out detect function.

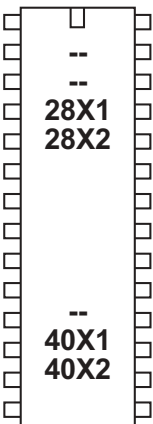
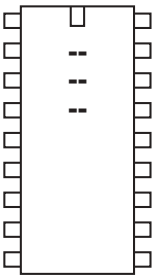
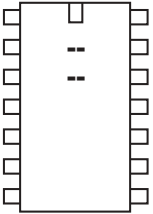
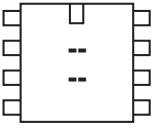
Information:

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
```



disconnect

Syntax:

DISCONNECT

Function:

Disconnect the PICAXE so that it does not scan for new downloads.

Information:

The PICAXE chips constantly scan the serial download pin to see if a computer is trying to initialise a new program download. However when it is desired to use the download pin for user serial communication (serrxd command), it is necessary to disable this scanning. Note that the serrxd command automatically includes a disconnect command.

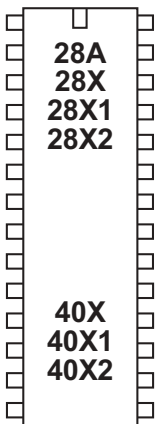
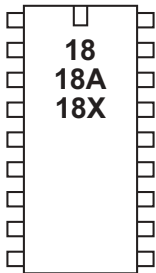
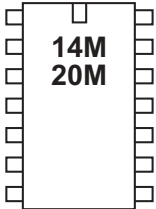
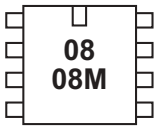
After disconnect is used it will not be possible to download a new program until:

- 1) the reconnect command is issued
- 2) a reset command is issued
- 3) a hardware reset is carried out

Remember that it is always possible to carry out a new download by carrying out the 'hard-reset' procedure.

Example:

```
serrxd [1000, timeout],@ptrinc,@ptrinc,@ptr
reconnect
```



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)

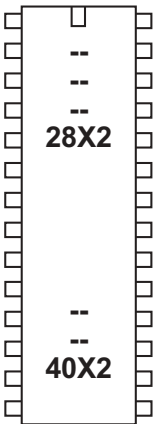
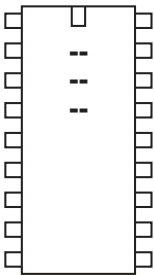
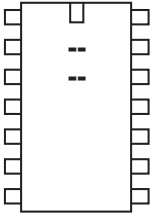
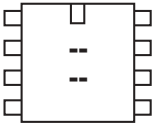
Information:

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:

```
do
  high 1
  pause 1000
  low 1
  pause 1000
  inc b1
  if pin1 = 1 then exit
loop while b1 < 5
```



doze

Syntax:

DOZE period

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

Function:

Doze 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 doze command puts the microcontroller into low power mode for a short period of time. Unlike the nap command, all timers are left on and so the pwmout, timer and servo commands will continue to function. 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.

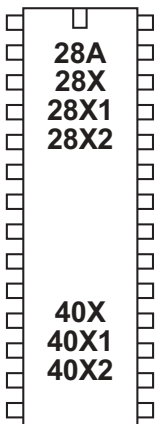
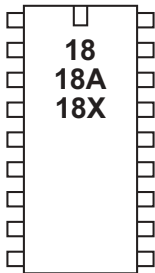
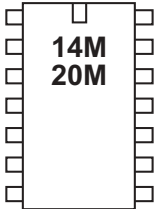
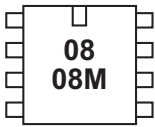
Period	Time Delay
0	18ms
1	32ms
2	72ms
3	144ms
4	288ms
5	576ms
6	1.152 s
7	2.304 s

Affect of increased clock speed:

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

Example:

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



eeeprom (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 microcontrollers data memory. The command does not affect program length.

With the PICAXE-08, 08M, 14M and 18 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-08	0 to (127 - number of used bytes)
PICAXE-08M	0 to (255 - number of used bytes)
PICAXE-14M / 20M	0 to (255 - number of used bytes)
PICAXE-18	0 to (127 - number of used bytes)

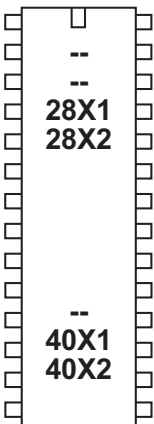
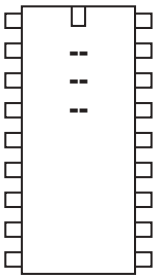
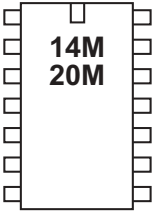
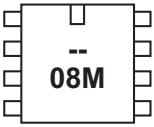
With the following microcontrollers the data memory is completely separate from the program and so no conflicts arise. The number of bytes available varies depending on microcontroller type as follows.

PICAXE-28, 28A	0 to 63
PICAXE-28X, 40X	0 to 127
PICAXE-18A, 18X, 28X1, 28X2, 40X1, 40X2	0 to 255

Example:

```
EEPROM 0,("Hello World")    ` save values in EEPROM

main:
  for b0 = 0 to 10           ` start a loop
    read b0,b1              ` read value from EEPROM
    serout 7,N2400,(b1)     ` transmit to serial LCD module
  next b0                   ` next character
```



enablebod

Syntax:

ENABLEBOD

Function:

Enable the on-chip brown out detect function.

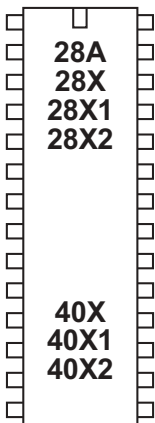
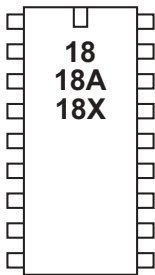
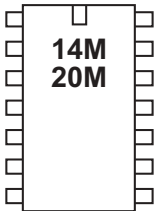
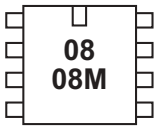
Information:

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.

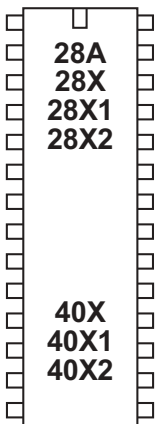
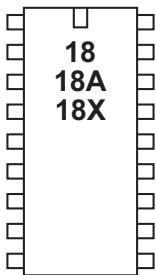
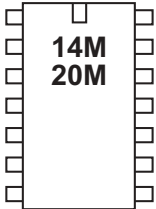
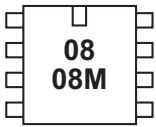
If you do not wish the end command to be carried out, place a 'stop' command at the bottom of the program. The stop command does not enter low power mode.

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:

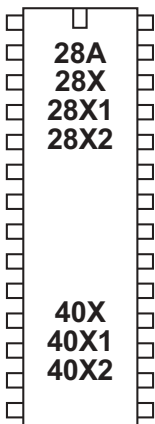
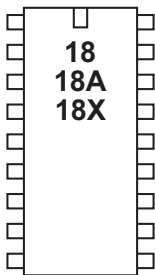
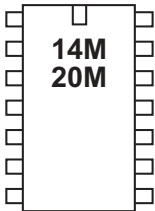
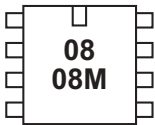
Exit is used to immediately terminate a do...loop or for...next program loop.

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).

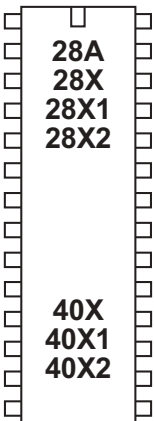
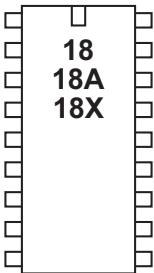
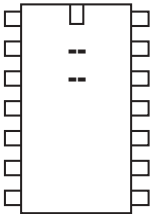
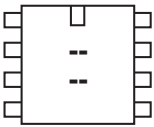
The for...next loop can be prematurely ended by use of the exit command.

Example:

main:

```
for b0 = 1 to 20 ` define loop for 20 times
  if pin1 = 1 then exit
  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

pause 2000       ` wait for 2 seconds
goto main        ` loop back to start
```



forward

Syntax:

FORWARD motor

- Motor is the motor name A or B.

Function:

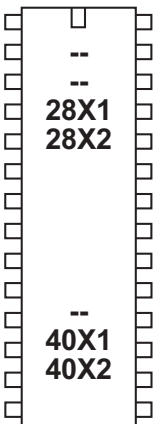
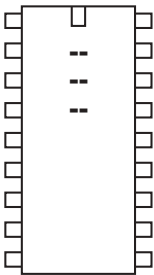
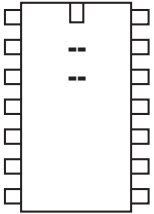
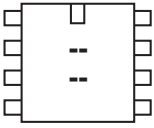
Make a motor output turn forwards

Information:

This is a 'pseudo' command designed for use by younger students with pre-assembled classroom models. It is actually equivalent to 'high 4 : low 5' (motor A) or 'high 6 : low 7' (motor B). This command is not normally used outside the classroom.

Example:

```
main:
    forward A           \ motor a on forwards
    wait 5              \ wait 5 seconds
    backward A         \ motor a on backwards
    wait 5              \ wait 5 seconds
    halt A              \ motor A reverse
    wait 5              \ wait 5 seconds
    goto main          \ loop back to start
```



get

Syntax:

GET location,variable,variable,WORD wordvariable...

- Location is a variable/constant specifying a register address. Valid values are 0 to 255 (0 to 127 for X1).
- 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 scratchpad.

Information:

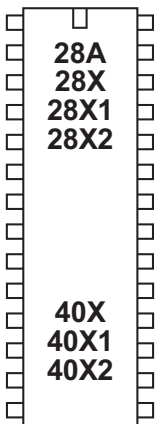
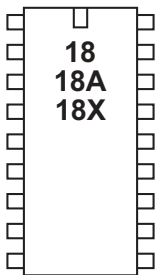
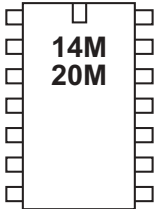
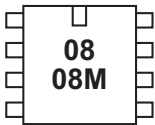
The function of the put/get commands is to store temporary byte data in the microcontrollers scratchpad memory. This allows the general purpose variables (b0, b1 etc) to be re-used in calculations.

Put and get have no affect on the scratchpad pointer and so the address next used by the indirect pointer (ptr) will not change during these commands.

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:

```
get 1,b1           ` put value of register 1 into variable b1
get 1, word w1
```



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.

The table shows the maximum number of gosubs available in each microcontroller . Gosubs can be nested 4 / 8 levels deep (ie there is a 4 / 8 level stack available in the microcontroller).

	gosubs	interrupt	stack
PICAXE-08	16	0	4
PICAXE-08M	15	1	4
PICAXE-14M / 20M	15	1	4
PICAXE-18	16	0	4
PICAXE-18A	15	1	4
PICAXE-18X	255	1	4
PICAXE-28A	15	1	4
PICAXE-28X / 40X	255	1	4
PICAXE-28X1 / 40X1	255	1	8
PICAXE-28X2 / 40X2	255	1	8

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 sub-procedure within variables, you can repeat a section of code from multiple places within the program. See the sample below for more information.

Example:

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

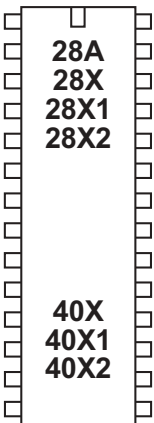
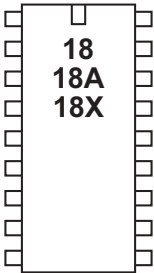
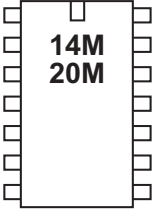
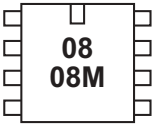
```

flsh:

```

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.

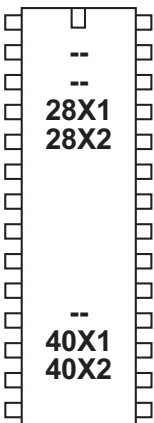
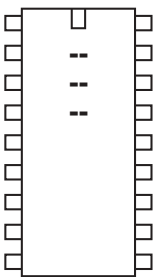
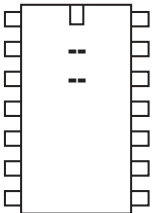
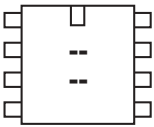
Information:

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

Example:

```
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
```





hi2cin

Syntax:

HI2CIN (variable,...)

HI2CIN location,(variable,...)

HI2CIN [newslave],(variable,...)

HI2CIN [newslave],location,(variable,...)

- Location is an optional variable/constant specifying a byte or word address.
- Variable(s) receives the data byte(s) read.
- Newslave is an optional new slave address for this (and all future) commands.

Function:

Read i2c location contents into variable(s).

Information:

Use of i2c parts is covered in more detail in the separate 'i2c Tutorial' datasheet.

This command is used to read byte data from an i2c device. Location defines the start address of the data read, although it is also possible to read more than one byte sequentially (if the i2c device supports sequential reads).

Location must be a byte or word as defined within the hi2csetup command. An hi2csetup command must have been issued before this command is used. The hi2csetup command sets the default slave address for this command. However when addressing multiple parts it may be necessary to repeatedly change the default slave address. This can be achieved via the optional [newslave] variable.

If the i2c hardware is incorrectly configured, or the wrong i2cslave data has been used, the value 255 (\$FF) will be loaded into each variable.

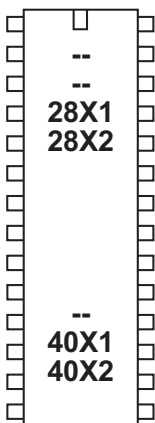
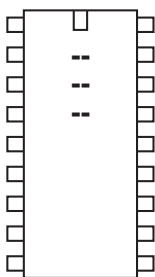
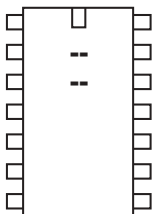
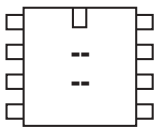
Example:

```
; Example of how to use DS1307 Time Clock
; Note the data is sent/received in BCD format.

' set PICAXE as master and DS1307 slave address
  hi2csetup i2cmaster, %11010000, i2cslow, i2cbyte

' read time and date and debug display

main:
  hi2cin 0, (b0,b1,b2,b3,b4,b5,b6,b7)
  debug b1
  pause 2000
  goto main
```

hi2cout

Syntax:

```
HI2COUT location,(variable,...)
```

```
HI2COUT (variable,...)
```

```
HI2COUT [newslave],location,(variable,...)
```

```
HI2COUT [newslave],(variable,...)
```

- Location is a variable/constant specifying a byte or word address.
- Variable(s) contains the data byte(s) to be written.
- Newslave is an optional new slave address for this (and all future) commands.

Function:

Write to i2c bus when acting as an i2c master device.

Information:

Use of i2c parts is covered in more detail in the separate 'i2c Tutorial' datasheet.

This command is used to write byte data to an i2c slave. Location defines the start address of the data to be written, although it is also possible to write more than one byte sequentially (if the i2c device supports sequential writes).

Location must be a byte or word as defined within the hi2csetup command. A hi2csetup command must have been issued before this command is used. The hi2csetup command sets the default slave address for this command. However when addressing multiple parts it may be necessary to repeatedly change the default slave address. This can be achieved via the optional [newslave] variable.

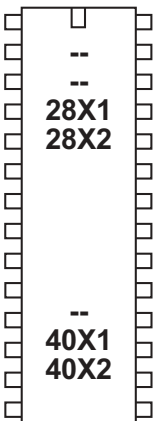
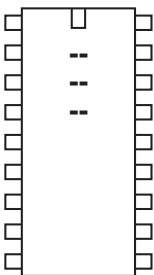
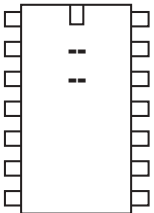
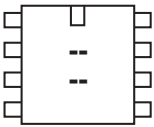
Example:

```
; Example of how to use DS1307 Time Clock
; Note the data is sent/received in BCD format.
; Note that seconds, mins etc are variables that need
; defining e.g. symbol seconds = b0 etc.

' set PICAXE as master and DS1307 slave address
  hi2csetup i2cmaster, %11010000, i2cslow, i2cbyte

'write time and date e.g. to 11:59:00 on Thurs 25/12/03
start_clock:
  let seconds = $00 ' 00 Note all BCD format
  let mins    = $59 ' 59 Note all BCD format
  let hour    = $11 ' 11 Note all BCD format
  let day     = $03 ' 03 Note all BCD format
  let date    = $25 ' 25 Note all BCD format
  let month   = $12 ' 12 Note all BCD format
  let year    = $03 ' 03 Note all BCD format
  let control = %00010000 ' Enable output at 1Hz

  hi2cout 0, (seconds,mins,hour,day,date,month,year,control)
end
```



hi2csetup

Syntax:

HI2CSETUP OFF

HI2CSETUP I2CSLAVE, slaveaddress

HI2CSETUP I2CMASTER, slaveaddress, mode, addresslen

- SlaveAddress is the i2c slave address
- Mode is the keyword `i2cfast` (1MHz) `i2cmedium` (400kHz) or `i2cslow` (100kHz)
- Addresslen is the keyword `i2cbyte` or `i2cword`. Note that this is the 'addressing method' used by the i2c device (i.e. some eeproms use a byte address, some use a word address). It is NOT the length of data returned by the `hi2cin` command, which is always a byte.

Function:

The `hi2csetup` command is used to configure the PICAXE pins for i2c use and to define the type of i2c device to be addressed.

Description:

Use of i2c parts is covered in more detail in the separate 'i2c Tutorial' datasheet.

hi2csetup - slave mode

Slave Address

The slave address is the address that is used by the PICAXE chip for identification. It can be a number between 0 and 127, but must be held in bits 7 to 1 of the address (not bits 6 - 0) e.g. `%1010000x`. Bit0 is the read/write bit and so ignored. If you are not sure which address to use we recommend the 'standard i2c EEPROM' address which is `%10100000`. Some special i2c addresses, such as 0, have special meanings under the i2c protocol (0 = broadcast all) and so are not recommended as they may cause unexpected behaviour on third party devices.

Description:

When in slave mode all i2c functions of the slave PICAXE chip are completely automatic. An i2c master can read or write to the slave PICAXE chip as if it was a 128 (X1) or 256 (X2) byte 24LCxx series EEPROM, with the scratchpad area acting as the memory transfer area. The master can read the slave PICAXE chip at any time. This does not have any effect on the slave PICAXE program.

However when the master writes to the slave PICAXE memory the 'hi2cflag' is set and the last address written to is saved in the 'hi2clast' variable. Therefore by polling the hi2cflag bit (or using `setintflags` to cause an interrupt) the PICAXE program can take action when a write has occurred. The hi2cflag must be cleared by the user program after use.

Example:

main:

```

if hi2cflag = 0 then main    ` poll flag

hi2cflag = 0                ` reset flag
get hi2clast,b1             ` get last byte written
let outpins = b1           ` set output pins
goto main

```

hi2csetup - master mode

If you are using a single slave i2c device alongside your PICAXE master you generally only need one hi2csetup command within a program. After the hi2csetup has been issued, hi2cin and hi2cout can be used to access the slave i2c device. When using multiple devices you can change the default slave address within the hi2cin or hi2cout command.

Slave Address

The slave address varies for different i2c devices (see table below). For the popular 24LCxx series serial EEPROMs the address is commonly %1010xxxx.

Note that some devices, e.g. 24LC16B, incorporate the block address (ie the memory page) into bits 1-3 of the slave address. Other devices include the external device select pins into these bits. In this case care must be made to ensure the hardware is configured correctly for the slave address used.

Bit 0 of the slave address is always the read/write bit. However the value entered using the i2cslave command is ignored by the PICAXE, as it is overwritten as appropriate when the slave address is used within the readi2c and writei2c commands.

Speed

Speed of the i2c bus can be selected by using one of the three keywords i2cfast, i2cmedium or i2cslow (1MHz, 400kHz or 100kHz). The internal slew rate control of the microcontroller is automatically enabled when required.

Address Length

i2c devices commonly have a single byte (i2cbyte) or double byte (i2cword) address. This must be correctly defined for the type of i2c device being used. If you use the wrong definition erratic behaviour will be experienced. When using the i2cword address length you must also ensure the 'address' used in the hi2cin and hi2cout commands is a word variable.

Settings for some common parts:

Device	Type	Slave	Speed	Address
24LC01B	EE 128	%1010xxxx	i2cfast	i2cbyte
24LC02B	EE 256	%1010xxxx	i2cfast	i2cbyte
24LC04B	EE 512	%1010xxbx	i2cfast	i2cbyte
24LC08B	EE 1kb	%1010xbbx	i2cfast	i2cbyte
24LC16B	EE 2kb	%1010bbbx	i2cfast	i2cbyte
24LC64	EE 8kb	%1010dddx	i2cfast	i2cword
24LC256	EE 64kb	%1010dddx	i2cfast	i2cword
DS1307	RTC	%1101000x	i2cslow	i2cbyte
MAX6953	5x7 LED	%101dddx	i2cfast	i2cbyte
AD5245	Digital Pot	%010110dx	i2cfast	i2cbyte
SRF08	Sonar	%1110000x	i2cfast	i2cbyte
AXE033	I2C LCD	\$C6	i2cslow	i2cbyte
CMPS03	Compass	%1100000x	i2cfast	i2cbyte
SPE030	Speech	%1100010x	i2cfast	i2cbyte

x = don't care (ignored)

b = block select (selects internal memory page within device)

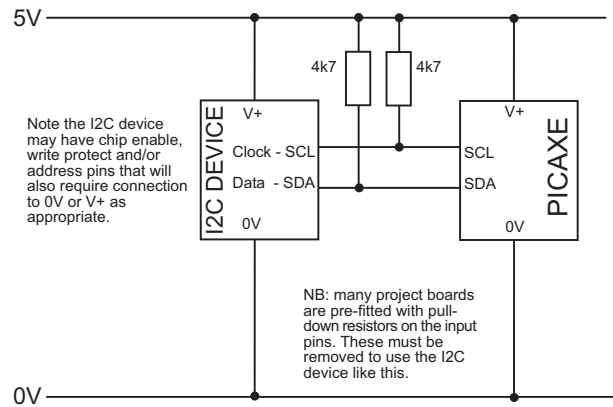
d = device select (selects device via external address pin polarity)

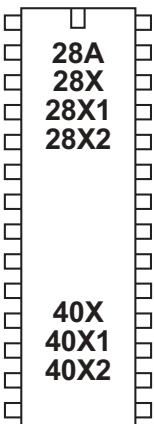
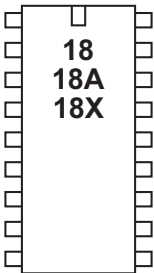
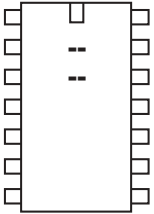
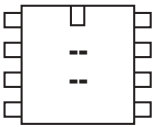
Affect of Increased Clock Speed:

Ensure you modify the mode keyword (`i2cfast_8`, `i2cslow_8`) at 8MHz or (`i2cfast_16`, `i2cslow_16`) at 16MHz for correct operation.

Advanced Technical Information:

Users familiar with assembler code programming may choose to create their own 'mode' settings to adjust the i2c communication speed. The mode value is a value between 0-127 that is the preload BRG value loaded into SSPADD. Bit 7 of the mode byte is used to set/clear the SSPSTAT,SMP slew control bit.





halt

Syntax:

HALT motor

- Motor is the motor name A or B.

Function:

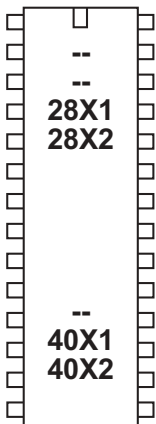
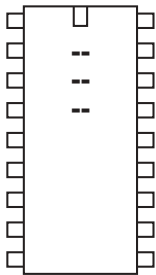
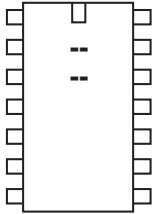
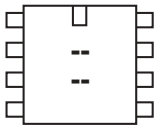
Make a motor output stop.

Information:

This is a 'pseudo' command designed for use by younger students with pre-assembled classroom models. It is actually equivalent to 'low 4 : low 5' (motor A) or 'low 6 : low 7' (motor B). This command is not normally used outside the classroom.

Example:

```
main: forward A      \ motor a on forwards
      wait 5         \ wait 5 seconds
      backward A    \ motor a on backwards
      wait 5         \ wait 5 seconds
      halt A        \ motor A reverse
      wait 5         \ wait 5 seconds
      goto main     \ loop back to start
```



hibernate

Syntax:

HIBERNATE config

- config is a constant/variable that sets the type of hibernation

Function:

Make the microcontroller sleep until a reset or interrupt occurs.

Information:

The hibernate command puts the microcontroller into very low power 'hibernation' mode. Unlike the sleep command, which wakes up every 2.3s, hibernate mode enters a state of permanent sleep. The only way to exit this deep sleep is via an external reset or via a hardware interrupt (hserin, hi2cin, etc.). A new program download from the computer will NOT wake the microcontroller.

For best low power performance, ensure any unused inputs are tied high/low, and that no outputs are being actively driven. The hibernate command automatically shuts down any on-board peripherals (timers, pwm etc) and disables the brown out detect circuit (equivalent of an automatic 'disable bod' command). If the brown out detect feature is not required after hibernate ends, the user program must disable it again via a disablebod command (as it is automatically re-enabled when the hibernate ends).

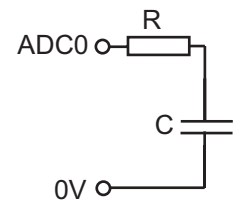
'config' value is used to disable/enable and set the 'ultra low power wake up feature' of analogue pin ADC0. A value of 0 disables this feature. When enabled, the hibernate will terminate after a capacitor (connected to ADC0) has discharged. This is more energy efficient than using the sleep command.

A non-zero config value enables the ULPWU feature on ADC0, and the actual config value sets the charging time (in ms) for the connected capacitor. Therefore the hibernate command first charges the capacitor, then hibernates, and then wakes up again once the capacitor has discharged.

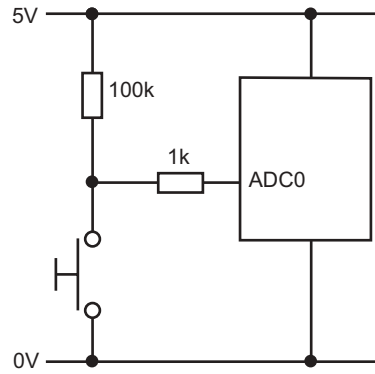
The discharge time is given by the following formula:

$$\text{Time} = \frac{(\text{initial C voltage} - 0.6) * C}{(\text{sink current} + \text{leakage current})}$$

The sink current is approximately 140nA with 5V power supply. Therefore the discharge time for a 200 ohm resistor and 1nF capacitor is approximately 30ms. This means the hibernate will end after approximately 30ms, although the discharge time is highly dependant on the capacitance (of the capacitor and circuit), and so, for example, long pcb tracks and moisture in the air can considerably affect these times.



MANUAL WAKEUP - The capacitor can also be completely replaced by a push-to-make switch (use 1k resistor as R and add another 100k resistor from the top of the switch to V+ to act as a positive voltage pull-up). The switch then acts as a manual 'wake-up' switch.

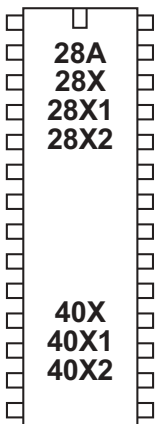
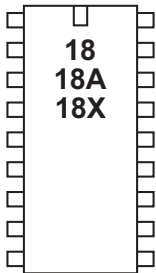
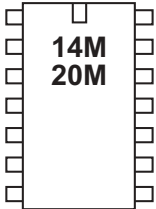
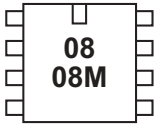


Note that the 1k is essential to prevent a possible short circuit situation (if the switch was pushed whilst the hibernate starts, as it will momentarily make ADC0 an output to 'charge the capacitor').

Example:

main:

```
toggle 1           ` toggle state of output 1
hibernate 50       ` hibernate after charging cap for 50ms
disablebod        ` turn bod off
goto main         ` loop back to start
```



high



Syntax:

HIGH pin {,pin,pin...}

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Make pin output high.

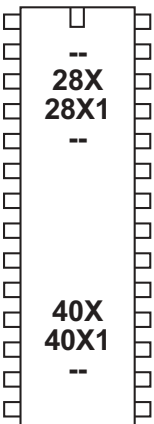
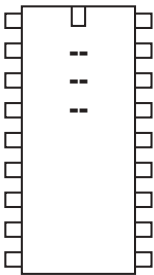
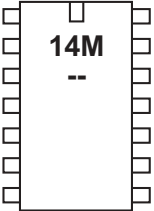
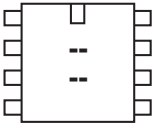
Information:

The high command switches an output on (high).

On microcontrollers with configurable input/output pins (e.g. PICAXE-08) 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
```

high portc

Syntax:

HIGH PORTC pin {,pin,pin...}

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

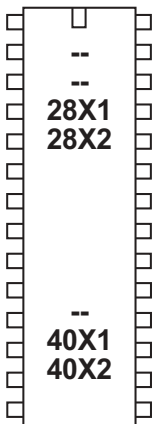
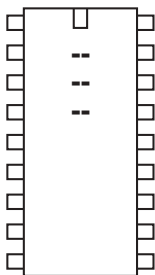
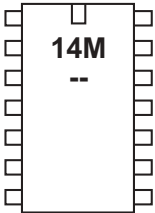
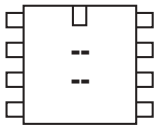
Make pin on portc output high.

Information:

The high command switches a portc output on (high).

Example:

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



hpwm

Syntax:

HPWM mode, polarity, setting, period, duty

HPWM OFF

- Mode is a variable/constant which specifies the hardware pwm mode
 - pwmsingle - 0
 - pwmhalf - 1
 - pwmfull_f - 2
 - pwmfull_r - 3
- polarity is a variable/constant which specifies the active polarity (DCBA)
 - pwmHHHH - 0
 - pwmLHLH - 1
 - pwmHLHL - 2
 - pwmLLLL - 3
- setting is a variable/constant which specifies a specific setting
 - single mode - bit mask %0000 to %1111 to dis/enable DCBA
 - half mode - dead band delay (value 0-127)
 - full mode - not used, enter 0 as default value
- Period is a variable/constant (0-255) which sets the PWM period (period is the length of 1 on/off cycle i.e. the total mark:space time).
- Duty is a variable/constant (0-1023) which sets the PWM duty cycle. (duty cycle is the mark or 'on time')

Function:

Hardware PWM is an advanced method of motor control using PWM methods. It can use a number of outputs and modes, as defined by the PIC microcontroller's internal pwm hardware.

hpwm can be used **instead of, not at the same time as**, the pwmout command on 2. However pwmout on 1 can be used simultaneously if desired.

Description:

hpwm gives access to the advanced pwm controller in the PIC microcontroller. It uses up to 4 pins, which are labelled here A,B,C,D for convenience.. Some of these pins normally 'default' to input status, in this case they will automatically be converted to outputs when the hpwm command is processed.

On 28/40 pin devices:

A is input 1 (C.1)
 B is output 2 (B.2)
 C is output 1 (B.1)
 D is output 4 (B.4)

On 14 pin devices:

A is input 2 (C.5)
 B is input 1 (C.4)
 C is input 0 (C.3)
 D is output 5 (C.2)

Not all pins are used in all hpwm modes. Unused bits are left as normal i/o pins.

single - A and/or B and/or C and/or D (each bit is selectable)
 half - A, B only
 full - A, B, C, D

The active polarity of each pair of pins can be selected by the polarity setting:

pwm_HHHH - A and C active high, B and D active high
 pwm_LHLH - A and C active high, B and D active low
 pwm_HLHL - A and C active low, B and D active high
 pwm_LLLL - A and C active low, B and D active low

When using active high outputs, it is important to use a pull-down resistor from the PICAXE pin (A-D) to 0V. When using active-low outputs a pull-up resistor is essential. The purpose of the pull-up/down resistor is to hold the FET driver in the correct state whilst the PICAXE chip initialises upon power up. During this short initialisation period the drivers are not actively driven (ie they 'float') and so the resistor is essential to hold the FET in the required off condition.

Single Mode (X1 and X2 parts only - not supported on the 14M)

In single mode each pin works independantly. It is therefore equivalent to a pwmout command. However more than one pin can be enabled at a time. Therefore this mode has two main uses:

- 1) To allow the equivalent of a 'pwmout' command on different outputs (than the pwmout command)
- 2) To allow pwmout on more than one pin (up to 4) at the same time. The pwmout applied to each output is identical. This is often used to provide a brightness control on multiple LEDs or to control multiple motors.

To enable a single output simply set its corresponding bit to '1' (D-C-B-A) within the settings byte of the command e.g. to enable all 4 pins use %1111

Half Mode (all parts)

In half mode outputs A and C control a half bridge. C and D are not used. The PWM signal is output on pin A, while the complementary PWM signal is output on pin B. The dead band delay 'setting' value is a very important value, without a correct value a shoot-through current may destroy the half bridge setup. This delay prevents both outputs being active at the same time. The command delay value (0-127) gives a delay equivalent to (value x oscillator speed (e.g. 4MHz) / 4). The value depends on the switch on/off characteristics of the FET drivers used.

See the hpwm motor driver datasheet for more details.

Full Mode (all parts)

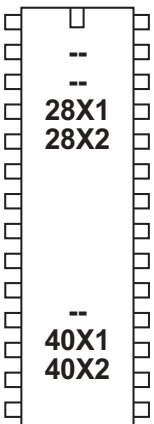
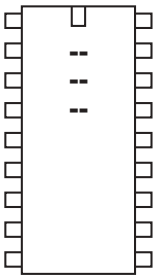
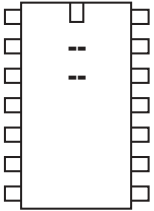
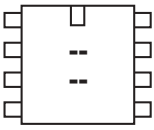
In full bridge mode outputs A, B, C and D control a full bridge.

In forward mode A is driven to its active state whilst D is modulated. B and C are in their inactive state.

In reverse mode C is driven to its active state whilst B is modulated. A and D are in their inactive state.

In this mode a deadband delay is generally not required as only one output is modulated at one time. However there can be conditions (when near near 100% duty cycle) where current shoot-through could occur. In this case it is recommended to either 1) switch off pwm before changing directions or 2) use a specialist FET driver that can switch the FET on quicker than it switches off (the opposite is normally true on non-specialist parts).

See the hpwm motor driver datasheet for more details.



hpwmduty

Syntax:

HPWMDUTY duty cycles

- Duty is a variable/constant (0-1023) which sets the PWM duty cycle. (duty cycle is the mark or 'on time')

Function:

Alter the duty cycle after a hpwm command has been issued.

Information:

On X1 and X2 parts the hpwmduty command can be used to alter the hpwm duty cycle without resetting the internal timer (as occurs with a hpwm command). A hpwm command must be issued before this command will function.

Information:

See the hpwm command for more details.

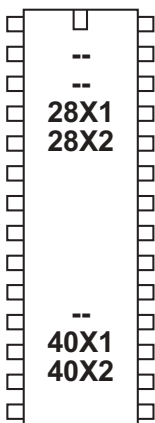
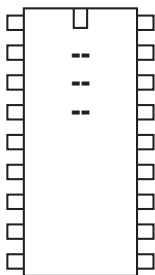
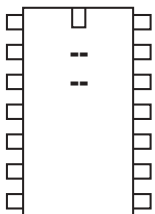
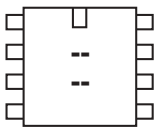
Example:

init:

```
hpwm 0,0,%1111,150,100 ' start pwm
```

main:

```
hpwmduty 150      ' set pwm duty
pause 1000        ' pause 1 s
hpwmduty 50       ' set pwm duty
pause 1000        ' pause 1 s
goto main         ' loop back to start
```



hserin

Syntax:

HSERIN spaddress, count {,(qualifier)}

HSERIN [timeout, address], spaddress, count {,(qualifier)}

- Qualifier is an optional single variable/constant (0-255) which must be received before subsequent bytes can be received and stored in scratchpad
- Spaddress is the first scratchpad address where bytes are to be received
- Count is the number of bytes to receive
- Timeout is an optional variables/constants which sets the timeout period in milliseconds
- Address is a label which specifies where to go if a timeout occurs.

Function:

Serial input via the hardware serial input pin (format 8 data, no parity, 1 stop).

Information:

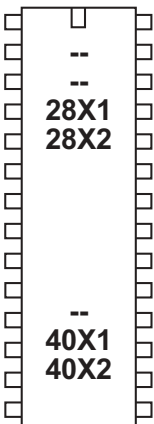
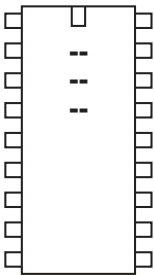
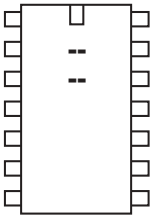
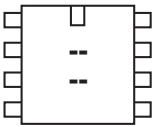
The hserin command is used to receive serial data from the fixed hardware serial input pin of the microcontroller. It cannot be used with the serial download input pin - use the serrxd command in this case.

Baud rate is defined by the hsersetup command, which must be issued before this command can be used. Note that you may prefer to background receive the serial data into the scratchpad (hence not requiring use of this command at all) - see the hsersetup command for more details (hserin only accepts data when the command is being processed - background receive accepts data all the time).

Users familiar with the serin command will note the hserin command has a completely different format. This is because the hserin command supports much higher baud rates than serin, and so is unable to process received bytes 'on the fly' (e.g. by changing ascii into binary, as with the serin # prefix), as there is insufficient time for this processing to occur before the next hserin byte is received (at high baud rates). Therefore the raw data is simply saved in the scratchpad memory area and the user program must then process the raw data when all the bytes have been received (e.g using the asciitobin command).

Example:

```
hsersetup B19200_16, %00 ` baud 19200 at 16MHz
main:
hserin [1000,main],0,4 ` receive 4 bytes into sp
ptr = 0 ` reset sp pointer
hserout 0,(@ptrinc,@ptrinc,@ptrinc,@ptr) ` echo out
goto main ` loop
```



hserout

Syntax:

HSEROUT break, ({#}data,{#}data...)

- Break is a variable/constant (0 or 1) which indicates whether to send a 'break' (wake-up) signal before the data is sent.
- 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 via the hardware serial output pin (8 data bits, no parity, 1 stop bit).

Information:

The hserout command is used to transmit serial data from the fixed hardware serial output pin of the microcontroller. It cannot be used with the serial download output pin - use the sertxd command in this case.

Polarity and baud rate are defined by the hsersetup command, which must be issued before this command can be used.

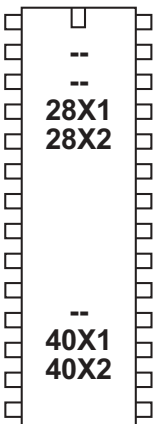
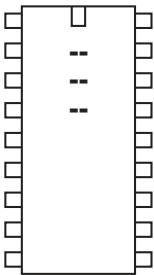
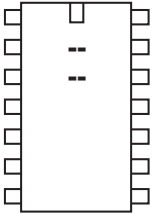
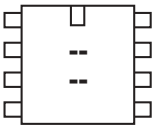
The # symbol allows ascii output. Therefore #b1, when b1 contains the data 126, will output the ascii characters "1" "2" "6" rather than the raw data byte '126'.

Example:

```

hsersetup B2400_4, %10
main:
  for b0 = 0 to 63          \ start a loop
    read b0,b1             \ read value into b1
    hserout 0,(b1)         \ transmit value to serial LCD
  next b0                  \ next loop

```



hsersetup

Syntax:

HSERSETUP OFF

HSERSETUP baud, mode

- Baud is a variable/constant which specifies the baud rate:

B300_X	where X =
B600_X	4 for 4MHZ
B1200_X	8 for 8 MHZ
B2400_X	16 for 16MHZ
B4800_X	20 for 20MHZ
B9600_X	32 for 32MHx
B19200_X	40 for 40 MHZ
B38400_X	
B57600_X	
B115200_X	

- Mode is a variable/constant whose bits specify special functions:

- bit0 - background receive serial data to the scratchpad
- bit1 - invert serial output data (default is true)

Function:

Configure the hardware serial port for serial operation.

Information:

The hsersetup command is used to configure the fixed hardware serial port of the microcontroller. It configures two pins to be dedicated to hserin and hserout.

The baud rate is configured by the baud value. For convenience a number of predefined values are predefined (e.g. B9600_4 for baud rate of 9600,n,8,1 at 4MHz operation)

Hardware serial input can be configured in two ways:

- 1) via hserin command only (mode bit0 = 0)
- 2) automatic in the background (mode bit0 = 1)

In automatic mode the hardware serial input is fully automated. Serial data received by the hardware pin is saved into the scratchpad memory area as soon as it is received. Upon the hsersetup command the serial pointer (hserptr) is reset to 0. When a byte is received it is saved to this scratchpad address, the hserptr variable is incremented and the hserinflag flag is set (must be cleared by user software).

The hardware serial input polarity is always true, it cannot be inverted. This is a limitation of the internal microcontroller structure. Therefore a MAX232 type inverter is required for computer connections.

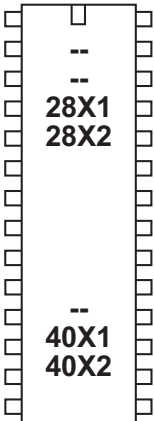
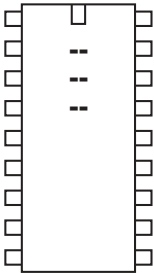
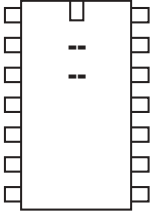
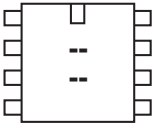
The hardware serial output polarity can be configured via bit1 of mode (0=true, 1=inverted). However, as the input polarity cannot ever be inverted it is often better to leave the output in the normal (true) state and use a MAX232 type inverter. However when interfacing to third party hardware (e.g. a serial LCD which only receives data) it may be easier to invert the data.

Example:

```
hsersetup B2400_4, %10
main:
  for b0 = 0 to 63           ` start a loop
    read b0,b1              ` read value into b1
    hserout 0,(b1)          ` transmit value to serial LCD
  next b0                   ` next loop
```

Advanced Technical Information:

Users familiar with assembler code programming may choose to create their own 'baud' rate settings. 'baud' must be a word value, and can be calculated from the PIC microcontroller datasheet equations when SYNC=0, BRGH = 1 and BRG16 = 1. It is not possible to change these settings.



hspiin (hshin)

Syntax:

HSPiIN (variable, {,variable,...})

- Variable receives the data.

Function:

The hspiin (hshin also accepted by the compiler) command shifts in a data byte using the SPI hardware pins.

Description:

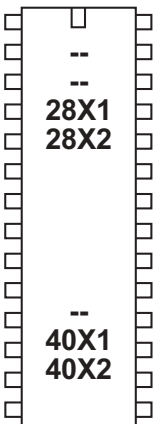
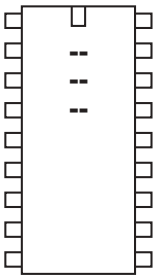
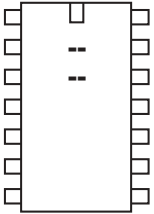
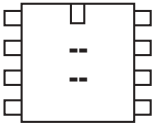
This command receives SPI data via the microcontroller's SPI hardware pins. This method is faster and more code efficient than using the 'bit-banged' spiin command.

When connecting SPI devices (e.g. EEPROM) remember that the data-in of the EEPROM connects to the data-out of the PICAXE, and vice versa.

Note that a hspissetup command must be issued before this command will function.

Example:

See the hspissetup command for a detailed example.



hspiout (hshout)

Syntax:

HSPiOUT (data, {,data,...})

- Data is a constant/variable of the byte data to output

Function:

The hspiout (hshout also accepted by the compiler) command shifts out data byte using the SPI hardware pins.

Description:

This command transmits SPI data via the microcontroller's SPI hardware pins. This method is faster and more code efficient than using the 'bit-banged' spiout command.

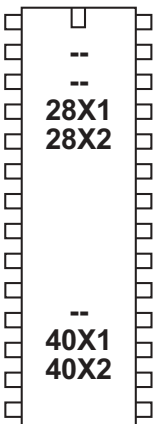
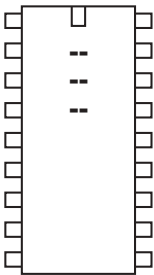
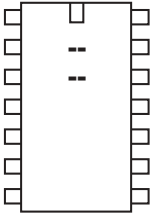
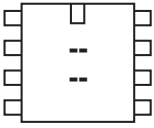
When connecting SPI devices (e.g. EEPROM) remember that the data-in of the EEPROM connects to the data-out of the PICAXE, and vice versa.

Note that a hspisepup command must be issued before this command will function.

Due to the internal operation of the microcontrollers SPI port, a hspiout command will only function when the hspiin 'input pin' is in the expected default state. If this pin is incorrect (e.g. high when it should be low), the hspiout byte cannot be sent (as the microcontroller automatically detects an SPI error condition). After 2.3 seconds of fault condition the PICAXE microcontroller will automatically reset.

Example:

See the hspisepup command for a detailed example.



hspissetup

Syntax:

HSPISSETUP OFF

HSPISSETUP mode, spispeed

- Mode is a constant/variable to define the mode
 - spimode00 (mode 0,0 - input sampled at middle of data time)
 - spimode01 (mode 0,1 - input sampled at middle of data time)
 - spimode10 (mode 1,0 - input sampled at middle of data time)
 - spimode11 (mode 1,1 - input sampled at middle of data time)
 - spimode00e (mode 0,0 - input sampled at end of data time)
 - spimode01e (mode 0,1 - input sampled at end of data time)
 - spimode10e (mode 1,0 - input sampled at end of data time)
 - spimode11e (mode 1,1 - input sampled at end of data time)
- Spispeed is a constant/variable to define the clock speed
 - spifast (clock freq / 4) (= 1MHz with 4MHz resonator)
 - spimedium (clock freq / 16) (= 250kHz with 4MHz resonator)
 - spislow (clock freq / 64) (= 63 kHz with 4MHz resonator)

Function:

The hspissetup command sets the microcontroller's hardware pins to SPI mode.

Description:

This command setups the microcontroller for SPI transmission via the microcontroller's SPI hardware pins. This method is faster and more code efficient than using the 'bit-banged' spiout (shiftout) command.

When connecting SPI devices (e.g. EEPROM) remember that the data-in (SDI) of the EEPROM connects to the data-out (SDO) of the PICAXE, and vice versa.

Advanced Technical Information:

Users familiar with assembler code programming may find the following microcontroller information useful:

spimode00	(CKP=0, CKE=1, SMP=0)
spimode01	(CKP=0, CKE=0, SMP=0)
spimode10	(CKP=1, CKE=1, SMP=0)
spimode11	(CKP=1, CKE=0, SMP=0)
spimode00e	(CKP=0, CKE=1, SMP=1)
spimode01e	(CKP=0, CKE=0, SMP=1)
spimode10e	(CKP=1, CKE=1, SMP=1)
spimode11e	(CKP=1, CKE=0, SMP=1)

Example:

This example shows how to read and write to a 25LC160 EEPROM.

Pin connection of the EEPROM is as follows:

1 - CS	picaxe output 7 (B.7)
2 - SO	picaxe input 4 (C.4)
3 - WP	+5V
4 - Vss	0V
5 - SI	picaxe input 5 (C.5)
6 - SCK	picaxe input 3 (C.3)
7 - HOLD	+5V
8 - Vdd	+5V

```
init:
    hspissetup spimodelle, spimedium    ` spi mode 1,1

    low cs                               ` enable chip select
    hspiout (6)                          ` send write enable
    high cs                               ` disable chip select

    low cs                               ` enable chip select
    hspiout (1,0)                        ` remove block protection
    high cs                               ` disable chip select
    pause 5                              ` wait write time

main:
    low cs                               ` enable chip select
    hspiout (6)                          ` send write enable
    high cs                               ` disable chip select

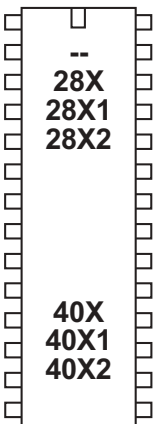
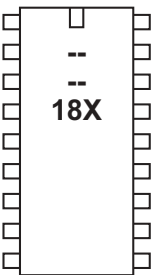
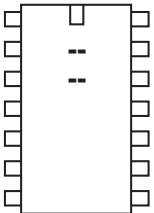
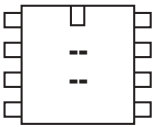
    low cs                               ` enable chip select
    hspiout (2,0,5,25)                  ` write 25 to address 5
    high cs                               ` disable chip select
    pause 5                              ` wait write time of 5ms

    low cs                               ` enable chip select
    hspiout (6)                          ` send write enable
    high cs                               ` disable chip select

    low cs                               ` enable chip select
    hspiout (3,0,5)                     ` send read command, address 5
    hspiin (b1)                          ` shift in the data
    high cs                               ` disable chip select

    low cs                               ` enable chip select
    hspiout (4)                          ` send write disable
    high cs                               ` disable chip select

    debug
    pause 1000
    goto main
```



i2cslave

Syntax:

I2CSLAVE slaveaddress, mode, addresslen

- SlaveAddress is the i2c slave address
- Mode is the keyword i2cfast (400kHz) or i2cslow (100kHz) at 4Mhz
- Addresslen is the keyword i2cbyte or i2cword

Function:

The i2cslave command is used to configure the PICAXE pins for i2c use (in MASTER mode) and to define the type of i2c device to be addressed.

For X1 and X2 parts the hi2csetup command is recommended instead.

Description:

Use of i2c parts is covered in more detail in the separate 'i2c Tutorial' datasheet.

If you are using a single i2c device you generally only need one i2cslave command within a program. With the PICAXE-18X device you should issue the command at the start of the program to configure the SDA and SCL pins as inputs to conserve power.

After the i2cslave has been issued, readi2c and writei2c can be used to access the i2c device.

Slave Address

The slave address varies for different i2c devices (see table below). For the popular 24LCxx series serial EEPROMs the address is commonly %1010xxxx.

Note that some devices, e.g. 24LC16B, incorporate the block address (ie the memory page) into bits 1-3 of the slave address. Other devices include the external device select pins into these bits. In this case care must be made to ensure the hardware is configured correctly for the slave address used.

Bit 0 of the slave address is always the read/write bit. However the value entered using the i2cslave command is ignored by the PICAXE, as it is overwritten as appropriate when the slave address is used within the readi2c and writei2c commands.

Mode

Speed mode of the i2c bus can be selected by using one of the two keywords i2cfast or i2cslow (400kHz or 100kHz). The internal slew rate control of the microcontroller is automatically enabled at the 400kHz speed (28X/40X). Note that the 18X internal architecture means that the slower speed is always used with the 18X, as it is not capable of processing at the faster speed.

Affect of Increased Clock Speed:

Ensure you modify the speed keyword (i2cfast_8, i2cslow_8) at 8MHz or (i2cfast_16, i2cslow_16) at 16MHz for correct operation.

Address Length

i2c devices commonly have a single byte (i2cbyte) or double byte (i2cword) address. This must be correctly defined for the type of i2c device being used. If you use the wrong definition erratic behaviour will be experienced.

When using the i2cword address length you must also ensure the 'address' used in the readi2c and writei2c commands is a word variable.

Settings for some common parts:

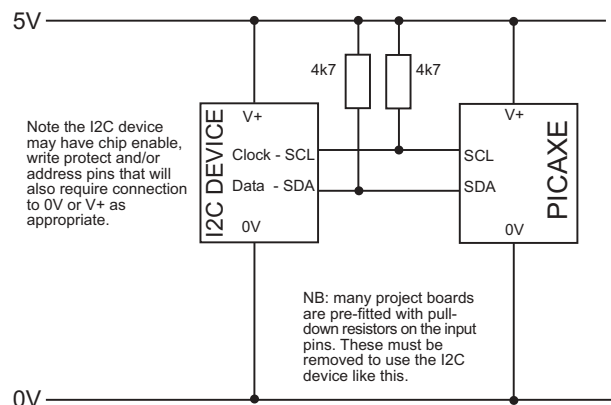
Device	Type	Slave	Speed	Mode
24LC01B	EE 128	%1010xxxx	i2cfast	i2cbyte
24LC02B	EE 256	%1010xxxx	i2cfast	i2cbyte
24LC04B	EE 512	%1010xxbx	i2cfast	i2cbyte
24LC08B	EE 1kb	%1010xbbx	i2cfast	i2cbyte
24LC16B	EE 2kb	%1010bbbx	i2cfast	i2cbyte
24LC64	EE 8kb	%1010dddx	i2cfast	i2cword
24LC256	EE 64kb	%1010dddx	i2cfast	i2cword
DS1307	RTC	%1101000x	i2cslow	i2cbyte
MAX6953	5x7 LED	%101dddx	i2cfast	i2cbyte
AD5245	Digital Pot	%010110dx	i2cfast	i2cbyte
SRF08	Sonar	%1110000x	i2cfast	i2cbyte
AXE033	I2C LCD	\$C6	i2cslow	i2cbyte
CMPS03	Compass	%1100000x	i2cfast	i2cbyte
SPE030	Speech	%1100010x	i2cfast	i2cbyte

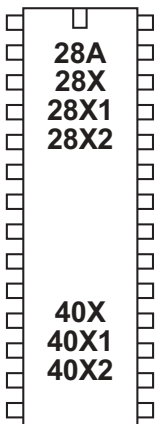
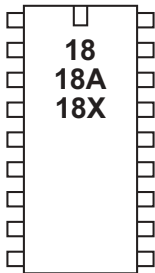
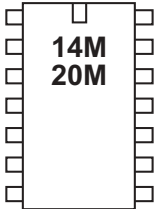
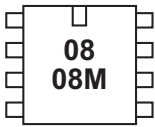
x = don't care (ignored)

b = block select (selects internal memory page within device)

d = device select (selects device via external address pin polarity)

See readi2c or writei2c for example program for DS1307 real time clock.





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/constant.

?? 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      `(single line structure)
if b0 > 1 then gosub label      `(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 else goto label2
```

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

```
if b0 > 1 then goto label : else : goto label2
```

or

```
if b0 > 1 then : goto label : else : goto label2
```

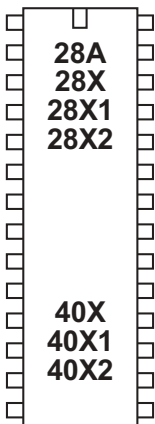
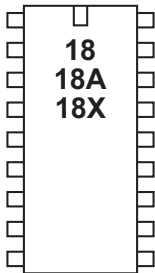
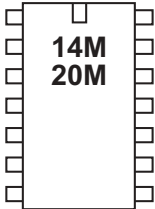
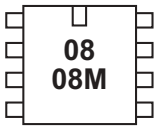
To achieve this structure the line must be re-written as

```
if b0 > 1 then
    goto label
else
    goto label2
endif
```

or

```
if b0 > 1 then : goto label : else : goto label2 : 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.

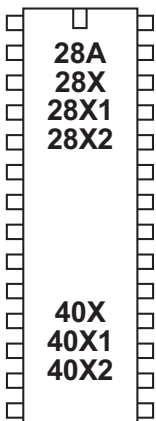
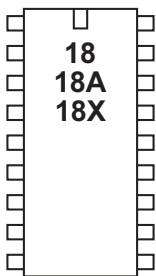
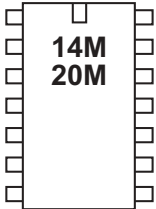
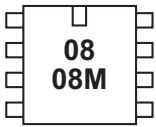
Examples:

Checking an input within a loop.

```

main:
    if pin0 = 1 then flsh ` jump to flsh if pin0 is high
    goto main           ` else loop back to start

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

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

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

- Variable(s) will be compared to value(s).
- Value is a variable/constant.

?? 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 exit a do...loop or for...next loop

Information:

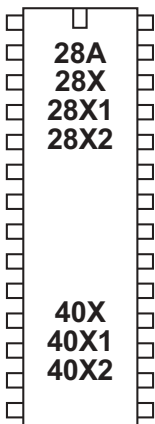
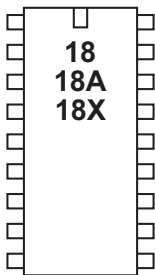
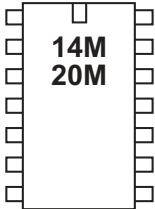
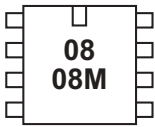
The if...then exit command is used to test input pin variables (or general variables) for certain conditions. If these conditions are met the current loop (do...loop or for...next) is prematurely ended.

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.

Example:

Checking an input within a do loop.

```
do
    if pin0 = 1 then exit ` sub to flsh if pin0 is high
loop
```



if...then gosub

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

Syntax:

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

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

?? 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 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.) i.e. 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:

Checking an input within a loop.

```
main:
  if pin0 = 1 then gosub flsh  \ sub to flsh if pin0 is high
  goto main                    \ else loop back to start

flsh: high 1                    \ switch on output 1
      pause 5000                \ wait 5 seconds
      low 1                      \ switch off output
      return
```

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

```
if pins = %10101010 then gosub label
```

To read the whole input port and mask individual inputs (e.g. 6 and 7)

```
let b1 = pins & %11000000
```

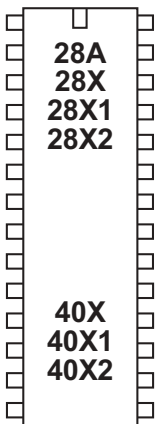
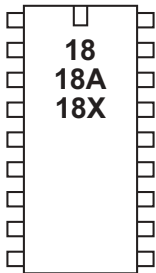
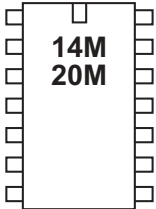
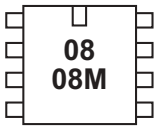
```
if b1 = %11000000 then gosub label
```

The words is (=), on (1) and off (0) can also be used with younger students.

loop1:

```
if pin0 is on then gosub flsh ` flsh if pin0 is high
goto loop1          ` else loop back to start
```

```
flsh: high 1          ` switch on output 1
pause 5000          ` wait 5 seconds
low 1                ` switch off output 1
return              ` return
```



inc

Syntax:

INC var

- var is the variable to increment

Function:

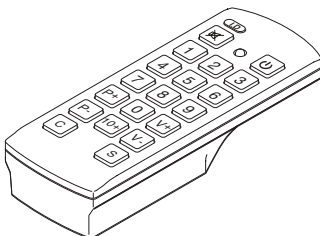
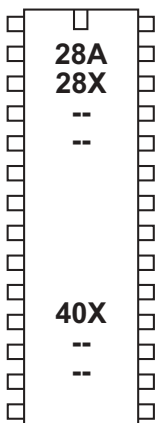
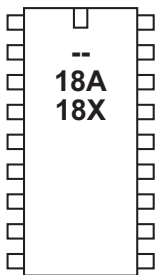
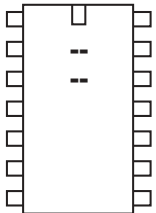
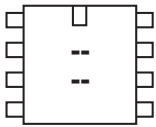
Increment (add 1 to) the variable value.

Information:

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

Example:

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



infrain

Syntax:
INFRAIN

Function:
Wait until a new infrared command is received.
See the *irin* command for X1 and X2 parts.

Description:
This command is primarily used to wait for a new infrared signal from the infrared TV style transmitter. It can also be used with an infraout signal from a separate PICAXE chip. All processing stops until the new command is received. The value of the command received is placed in the predefined variable 'infra'.

The infra-red input is input 0 on all parts that support this command.
The variable 'infra' is separate from the other byte variables.

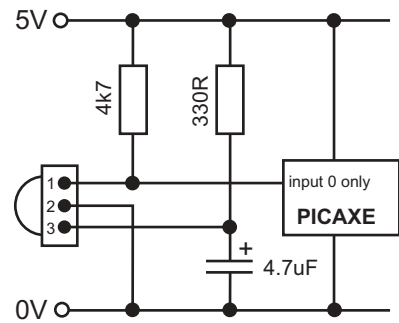
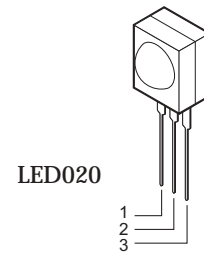
After using this command you may have to perform a 'hard reset' to download a new program to the microcontroller. See the Serial Download section for more details.

Affect of Increased Clock Speed:
This command will only function at 4MHz

Use of TVR010 Infrared Remote Control:
The table shows the value that will be placed into the variable 'infra' depending on which key is pressed on the transmitter.

Before use (or after changing batteries) the TVR010 transmitter must be programmed with 'Sony' codes as follows:

1. Insert 3 AAA size batteries, preferably alkaline.
2. Press 'C'. The LED should light.
3. Press '2'. The LED should flash.
4. Press '1'. The LED should flash.
5. Press '2'. The LED should flash and then go out.



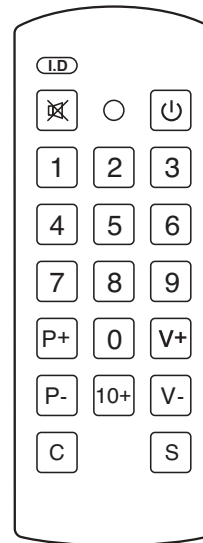
Key	Value
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
P+	10
0	11
V+	12
P-	13
10+	14
V-	15
Mute	16
Power	17

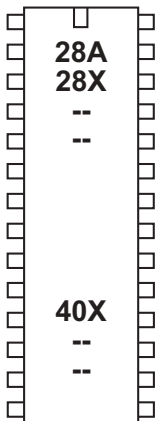
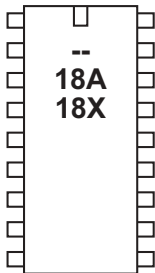
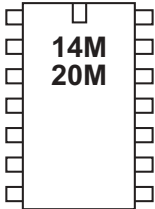
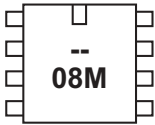
Example:

```

main:
  infrain
  if infra = 1 then swon1
  if infra = 2 then swon2
  if infra = 3 then swon3
  if infra = 4 then swoff1
  if infra = 5 then swoff2
  if infra = 6 then swoff3
  goto main

swon1:    high 1
          goto main
swon2:    high 2
          goto main
swon3:    high 3
          goto main
swoff1:   low 1
          goto main
swoff2:   low 2
          goto main
swoff3:   low 3
          goto main
  
```





infrain2

Syntax:
INFRAIN2

Function:
Wait until a new infrared command is received.
See the *irin* command for X1 and X2 parts.

Description:
This command is used to wait for an infraout signal from a separate PICAXE chip. It can also be used with an infrared signal from the infrared TV style transmitter (i.e.. can replace *infrain*). 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/14M/20M '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 program to the microcontroller. See the Serial Download section for more details.

Affect of Increased Clock Speed:

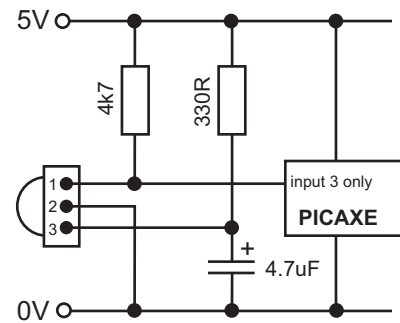
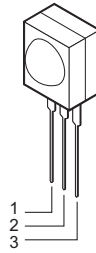
This command will only function at 4MHz. Use a *setfreq m4* command before this command if using 8MHz speed,

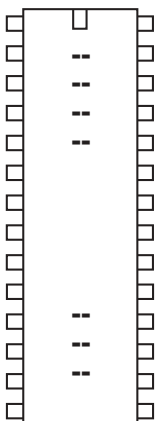
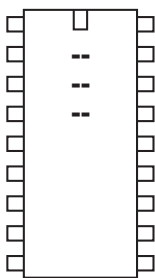
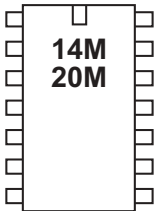
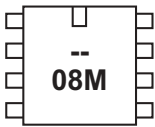
Example:

```
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.

See the *irout* command for X1 and X2 parts.

Description:

This command is used to transmit the infra-red data to Sony™ device (can also be used to transmit data to another PICAXE 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.

device - 5 bit device ID (0-31)

data - 7 bit data (0-127)

When using this command to transmit data to another PICAXE the device ID used must be value 1 (TV). The *infraout* command can be used to transmit any of the valid TV command 0-127. Note that the Sony protocol only uses 7 bits for data, and so data of value 128 to 255 is not valid.

Therefore the valid *infraout* command for use with *infrain2* is

infraout 1,x (where x = 0 to 127)

Sony SIRC protocol:

Start	Data0	Data1	Data2	Data3	Data4	Data5	Data6	ID0	ID1	ID2	ID3	ID4
2.4ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms

The SIRC protocol uses a 38kHz modulated infra-red signal consisting of a start bit (2.4ms) followed by 12 data bits (7 data bits and 5 device ID bits). Logic level 1 is transmitted as a 1.2 ms pulse, logic 0 as a 0.6ms pulse. Each bit is separated by a 0.6ms silence period.

Example:

All commercial remote controls repeat the signal every 45ms whilst the button is held down. Therefore when using the PICAXE system higher reliability may be gained by repeating the transmission (e.g. 10 times) within a *for..next* loop.

```
for b1 = 1 to 10
  infraout 1,5
  pause 45
next b1
```

Interaction between infrain, infrain2 and infraout command.

Infrain and Infraout

The original infrain command was designed to react to signals from the TV style remote control TVR010. Therefore it only acknowledges the data sent from the 17 buttons on this remote (1-9, 0, 10+, P+, P-, V+, V-, MUTE, PWR) with a value between 1 and 17.

The infraout command can be used to 'emulate' the TVR010 remote to transmit signals that will be acceptable for the infrain command. The values to be used for each TV remote button are shown in the table.

TVR010 TV Remote Control	infraout equivalent command	infrain variable data value	infrain2 variable data value
1	infraout 1,0	1	0
2	infraout 1,1	2	1
3	infraout 1,2	3	2
4	infraout 1,3	4	3
5	infraout 1,4	5	4
6	infraout 1,5	6	5
7	infraout 1,6	7	6
8	infraout 1,7	8	7
9	infraout 1,8	9	8
P+	infraout 1,16	10	16
0	infraout 1,9	11	9
V+	infraout 1,18	12	18
P-	infraout 1,17	13	17
10+	infraout 1,12	14	12
V-	infraout 1,19	15	19
MUTE	infraout 1,20	16	20
PWR	infraout 1,21	17	21

Infrain2 and Infraout

The infrain2 command will react to *any* of the valid TV data commands (0 to 127).

The infraout command can be used to transmit any of the valid TV command 0-127. Note that the Sony protocol only uses 7 bits for data, and so data of 128 to 255 is not valid.

Therefore the valid infraout command for use with infrain2 is (where x = 0 to 127)

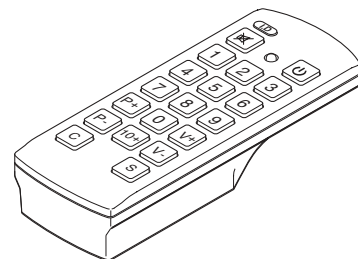
infraout 1,x

Affect of Increased Clock Speed:

This command will only function at 4MHz.

Common Sony Device IDs.:

TV	1	VTR3	11
VTR1	2	Surround Sound	12
Text	3	Audio	16
Widescreen	4	CD Player	17
MDP / Laserdisk	6	Pro-Logic	18
VTR2	7	DVD	26



Button infraout data for a typical Sony TV (device ID 1)

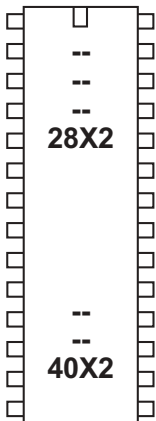
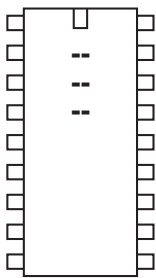
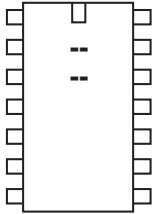
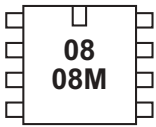
000	1 button
001	2 button
002	3 button
003	4 button
004	5 button
005	6 button
006	7 button
007	8 button
008	9 button
009	10 button/0 button
011	Enter
016	channel up
017	channel down
018	volume up
019	volume down
020	Mute
021	Power
022	Reset TV
023	Audio Mode:Mono/SAP/Stereo
024	Picture up
025	Picture down
026	Color up
027	Color down
030	Brightness up
031	Brightness down
032	Hue up
033	Hue down
034	Sharpness up
035	Sharpness down
036	Select TV tuner
038	Balance Left
039	Balance Right
041	Surround on/off
042	Aux/Ant
047	Power off
048	Time display
054	Sleep Timer
058	Channel Display
059	Channel jump
064	Select Input Video1
065	Select Input Video2
066	Select Input Video3

Button infraout data for a typical Sony TV (continued...)

- 074 Noise Reduction on/off
- 078 Cable/Broadcast
- 079 Notch Filter on/off
- 088 PIP channel up
- 089 PIP channel down
- 091 PIP on
- 092 Freeze screen
- 094 PIP position
- 095 PIP swap
- 096 Guide
- 097 Video setup
- 098 Audio setup
- 099 Exit setup
- 107 Auto Program
- 112 Treble up
- 113 Treble down
- 114 Bass up
- 115 Bass down
- 116 + key
- 117 - key
- 120 Add channel
- 121 Delete channel
- 125 Trinitone on/off
- 127 Displays a red RtestS on the screen

Button infraout data for a typical Sony VCR (device ID 2 or 7)

000	1 button
001	2 button
002	3 button
003	4 button
004	5 button
005	6 button
006	7 button
007	8 button
008	9 button
009	10 button/0 button
010	11 button
011	12 button
012	13 button
013	14 button
020	X 2 play w/sound
021	power
022	eject
023	L-CH/R-CH/Stereo
024	stop
025	pause
026	play
027	rewind
028	FF
029	record
032	pause engage
035	X 1/5 play
040	reverse visual scan
041	forward visual scan
042	TV/VTR
045	VTR from TV
047	power off
048	single frame reverse/slow reverse play
049	single frame advance/slow forward play
060	aux
070	counter reset
078	TV/VTR
083	index (scan)
106	edit play
107	mark



input

Syntax:

INPUT pin,pin,pin...

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Make pin an input.

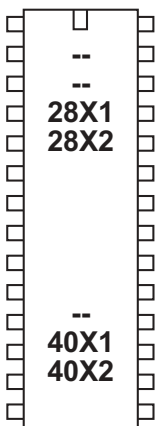
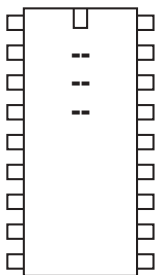
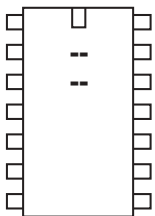
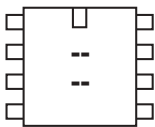
Information:

This command is only required on microcontrollers with programmable input/output pins (e.g. PICAXE-08M). 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 (apart from out0 on the PICAXE-08, which is always an output).

Example:

```
main:
    input 1          \ make pin input
    reverse 1       \ make pin output
    reverse 1       \ make pin input
    output 1        \ make pin output
```



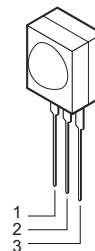
irin

Syntax:

IRIN pin, variable

IRIN [timeout, address], pin, variable

- Timeout is a variable/constant which sets the timeout period in milliseconds
- Address is a label which specifies where to go if a timeout occurs.
- pin is a variable/constant (0-7) which specifies the i/o pin to use.
- Variable receives the data

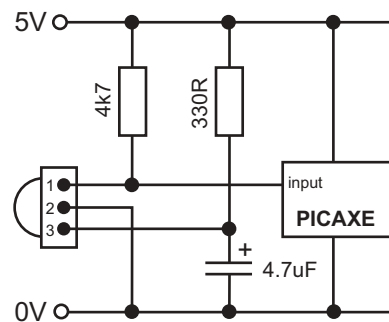


Function:

Wait until a new infrared command is received. This command is similar to the 'infrain2' command found on other PICAXE devices, but can be used on any input pin.

Description:

This command is used to wait for an infraout signal from a separate PICAXE chip. It can also be used with an infrared signal from the infrared TV style transmitter (i.e. can replace infrain). All processing stops until the new command is received, but after a timeout peiod program flow will jump to 'address'. The value of the command received is placed into the defined variable. 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



Start	Data0	Data1	Data2	Data3	Data4	Data5	Data6	ID0	ID1	ID2	ID3	ID4
2.4ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms	1.2 or 0.6ms

TVR010 remote control.

Affect of Increased Clock Speed:

This command will automatically use the internal 4MHz resonator for correct operation.

Example:

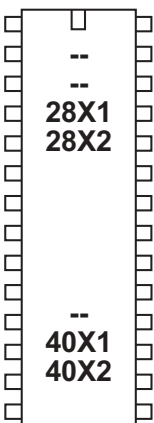
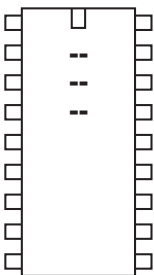
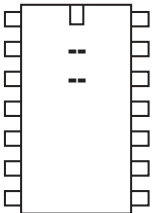
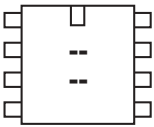
main:

```

irin [1000,main],3,b0      'wait for new signal
if b0 = 1 then swon1      'switch on 1
if b0 = 4 then swoff1     'switch off 1
goto main
    
```

```

swon1:    high 1
          goto main
swoff1:   low 1
          goto main
    
```



irout

Syntax:

IROUT pin,device,data

- pin is a variable/constant (0-7) which specifies the i/o pin to use.
- 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.

This command is similar to the 'infraout' command found on other PICAXE devices, but can be used on any output pin.

Description:

This command is used to transmit the infra-red data to Sony™ device (can also be used to transmit data to another PICAXE that is using the irin, infrain or infrain2 command). Data is transmitted via an infra-red LED using the SIRC (Sony Infra Red Control) protocol.

device - 5 bit device ID (0-31)

data - 7 bit data (0-127)

When using this command to transmit data to another PICAXE the device ID used must be value 1 (TV). The irout command can be used to transmit any of the valid TV command 0-127. Note that the Sony protocol only uses 7 bits for data, and so data of value 128 to 255 is not valid.

Therefore the valid infraout command for use with infrain2/infrain/irin is

irout 1,1,x (where x = 0 to 127)

Sony SIRC protocol:

The SIRC protocol uses a 38KHz modulated infra-red signal consisting of a start bit (2.4ms) followed by 12 data bits (7 data bits and 5 device ID bits). Logic level 1 is transmitted as a 1.2 ms pulse, logic 0 as a 0.6ms pulse. Each bit is separated by a 0.6ms silence period. For more information about the protocol see the 'infraout' command description.

Affect of Increased Clock Speed:

This command will automatically use the internal 4MHz resonator for correct operation.

Example:

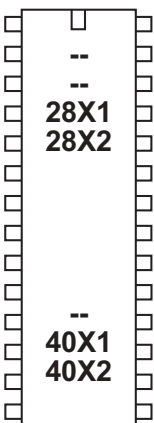
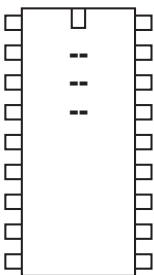
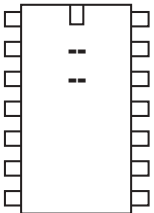
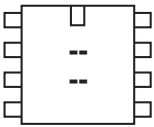
All commercial remote controls repeat the signal every 45ms whilst the button is held down. Therefore when using the PICAXE system higher reliability may be gained by repeating the transmission (e.g. 10 times) within a for..next loop.

for b1 = 1 to 10

irout 1,1,5

pause 45

next b1



kbin

Syntax:

KBIN variable

KBIN [timeout, address],variable

- Variable receives the key
- Timeout is a variable/constant which sets the timeout period in milliseconds
- Address is a label which specifies where to go if a timeout occurs.

Function:

Wait until a new keyboard press is received. This command is similar to the keyin command found on other PICAXE parts.

Information:

This command is used to wait for a new key press from a computer keyboard (connected directly to the PICAXE - not the keyboard used whilst programming, see keyed command for connection details). All processing stops until the new key press is received, but program flow will jump to address after the timeout period. The value of the key press received is placed in the variable.

Note the design of the keyboard means that the value of each key is not logical, each key value must be identified from the table (see keyin command). Some keys use two numbers, the first \$EO is ignored by the PICAXE and so keyvalue will return the second number. Note all the codes are in hex and so should be prefixed with \$ whilst programming. The PAUSE and PRNT SCRN keys cannot be used reliably as they have a special long multi-digit code. Also note that some keys may not work correctly when the 'Nums Lock' LED is set on with the keyed command.

The sample file 'keyin.bas' (installed in the \samples folder) provides details on how you can convert the key presses into ASCII characters by means of a look up table.

Affect of Increased Clock Speed:

This command will automatically use the internal 4MHz resonator for correct operation.

Example:

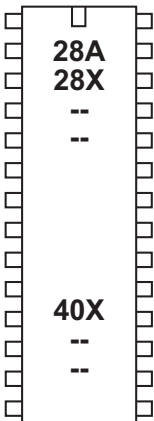
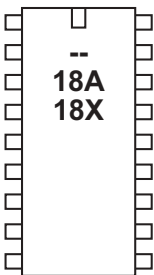
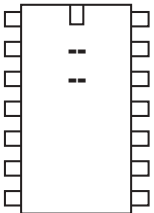
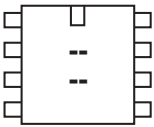
```

main:
    kbin [1000,main],b1           'wait for new signal
    if b1 = $45 then swon1       'switch on 1
    if b1= $25 then swoff1      'switch off 1
    goto main

swon1:    high 1
             goto main

swoff1:  low 1
             goto main

```



keyin

Syntax:

KEYIN

Function:

Wait until a new keyboard press is received.

See the *kbin* command for *X1* and *X2* parts.

Information:

This command is used to wait for a new key press from a computer keyboard (connected directly to the PICAXE - not the keyboard used whilst programming, see keyed command for connection details). All processing stops until the new key press is received. The value of the key press received is placed in the predefined variable 'keyvalue'.

Note the design of the keyboard means that the value of each key is not logical, each key value must be identified from the table on the next page. Some keys use two numbers, the first \$E0 is ignored by the PICAXE and so keyvalue will return the second number. Note all the codes are in hex and so should be prefixed with \$ whilst programming. The PAUSE and PRNT SCRN keys cannot be used reliably as they have a special long multi-digit code. Also note that some keys may not work correctly when the 'Nums Lock' LED is set on with the keyed command.

The sample file 'keyin.bas' (installed in the \samples folder) provides details on how you can convert the key presses into ASCII characters by means of a look up table.

After using this command you may have 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:

This command will only function at 4MHz.

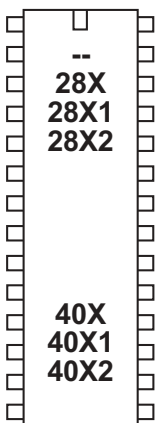
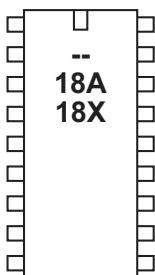
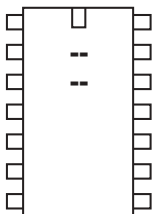
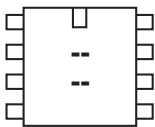
Example:

```
main:
    keyin                                'wait for new signal
    if keyvalue = $45 then swon1         'switch on 1
    if keyvalue = $25 then swoff1       'switch off 1
    goto main

swon1:    high 1
          goto main

swoff1:   low 1
          goto main
```

KEY	CODE	KEY	CODE	KEY	CODE
A	1C	9	46	[54
B	32	`	0E	INSERT	E0,70
C	21	-	4E	HOME	E0,6C
D	23	=	55	PG UP	E0,7D
E	24	\	5D	DELETE	E0,71
F	2B	BKSP	66	END	E0,69
G	34	SPACE	29	PG DN	E0,7A
H	33	TAB	0D	U ARROW	E0,75
I	43	CAPS	58	L ARROW	E0,6B
J	3B	L SHIFT	12	D ARROW	E0,72
K	42	L CTRL	14	R ARROW	E0,74
L	4B	L GUI	E0,1F	NUM	77
M	3A	L ALT	11	KP /	E0,4A
N	31	R SHFT	59	KP *	7C
O	44	R CTRL	E0,14	KP -	7B
P	4D	R GUI	E0,27	KP +	79
Q	15	R ALT	E0,11	KP EN	E0,5A
R	2D	APPS	E0,2F	KP .	71
S	1B	ENTER	5A	KP 0	70
T	2C	ESC	76	KP 1	69
U	3C	F1	05	KP 2	72
V	2A	F2	06	KP 3	7A
W	1D	F3	04	KP 4	6B
X	22	F4	06	KP 5	73
Y	35	F5	03	KP 6	74
Z	1A	F6	0B	KP 7	6C
0	45	F7	83	KP 8	75
1	16	F8	0A	KP 9	7D
2	1E	F9	01]	5B
3	26	F10	09	;	4C
4	25	F11	78	'	52
5	2E	F12	07	,	41
6	36	PRNT SCR	??	.	49
7	3D	SCROLL	7E	/	4A
8	3E	PAUSE	??		



keyled (kbled)

Syntax:

keyled mask

- Mask is a variable/constant which specifies the LEDs to use.

Function:

Set/clear the keyboard LEDs

Information:

This command is used to control the LEDs on a computer keyboard (connected directly to the PICAXE - not the keyboard used whilst programming). The mask value sets the operation of the LEDs.

Mask is used as follows:

- Bit 0 - Scroll Lock (1=on, 0=off)
- Bit 1 - Num Lock (1=on, 0=off)
- Bit 2 - Caps Lock (1=on, 0=off)
- Bit 3-6 - Not Used
- Bit 7 - Disable Flash (1=no flash, 0=flash)

On reset mask is set to 0, and so all three LEDs will flash when the 'keyin' command detects a new key hit. This provides the user with feedback that the key press has been detected by the PICAXE. This flashing can be disabled by setting bit 7 of mask high. In this case the condition of the three LEDs can be manually controlled by setting/clearing bits 2-0.

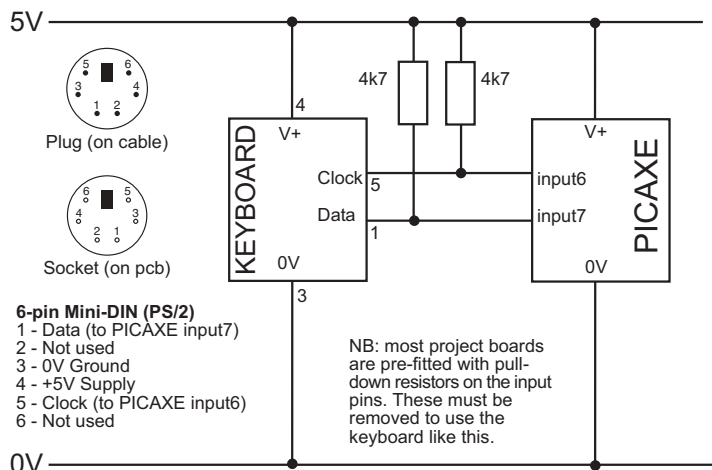
Affect of Increased Clock Speed:

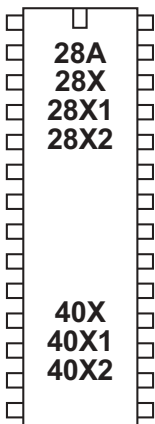
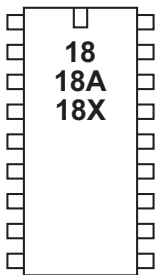
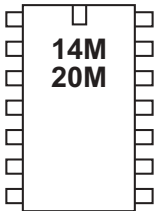
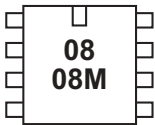
This command will only function at 4MHz.

Example:

main:

```
keyled %10000111 \ all LEDs on
pause 500 \ pause 0.5s
keyled %10000000 \ all LEDs off
pause 500 \ pause 0.5s
goto main \ loop
```



**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 65335. 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, 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 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 <i>not</i> the same as NAND)
ORNOT	/	; bitwise OR NOT	(NB this is <i>not</i> 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)

The X1 and X2 parts also support these unary commands

SIN	; sine of angle (0 to 65535) in degrees (value * 100 is returned)
COS	; cosine of angle in degrees (value * 100 is returned)
SQR	; square root
INV	; invert
NCD	; encoder (2n power encoder)
DCD	; decoder (2n power decoder)
BINTOBCD	; convert binary value to BCD
BCDTOBIN	; convert BCD value to binary
REV	; reverse a number of bits
DIG	; return a BCD digit

All mathematics is performed strictly from left to right.

On X1 and X2 parts it is possible to enclose part equations in brackets e.g.

```
let w1 = w2 / ( b5 + 2)
```

On all other chips it is not possible to enclose part equations in brackets e.g.

```
let w1 = w2 / ( b5 + 2)
```

is not valid. This would need to be entered as an equivalent e.g.

```
let w1 = b5 + 2
```

```
let w1 = w2 / w1
```

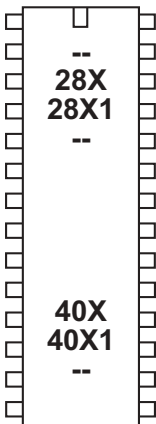
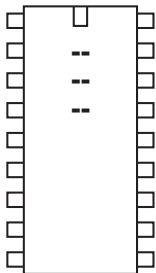
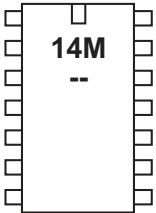
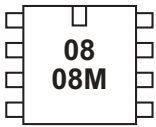
Further Information:

For further information please see the 'variable mathematics' section of this manual.

Example:

```
main:
    let b0 = b0 + 1           \ increment b0
    sound 7,(b0,50)         \ make a sound
    if b0 > 50 then rest    \ after 50 reset
    goto main               \ loop back to start

rest:
    let b0 = b0 max 10      \ limit b0 back to 10
                           \ as 10 is the maximum value
    goto main               \ loop back to start
```



let dirs =

let dirsc =

Syntax:

{LET} dirs = value

{LET} dirsc = value

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

Function:

Configure pins as inputs or outputs (let dirs =) (PICAXE-08/08M)

Configure pins as inputs or outputs on portc (let dirsc =) (PICAXE-14M)

Configure pins as inputs or outputs on portc (let dirsc =) (PICAXE-28X/40X)

Configure pins as inputs or outputs on portc (let dirsc =) (PICAXE-28X1/40X1)

Information:

Some microcontrollers allow inputs to be configured as inputs or outputs. In these cases it is necessary to tell the microcontroller which pins to use as inputs and/or outputs (all are configured as inputs on first power up). There are a number of ways of doing this:

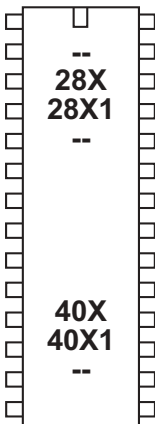
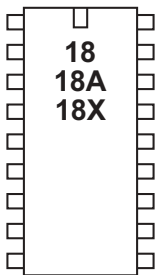
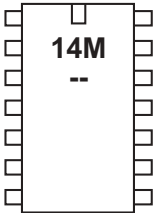
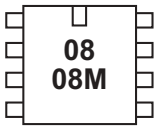
- 1) Use the input/output/reverse commands.
- 2) Use an output command (high, pulsout etc) that automatically configures the pin as an output.
- 3) Use the let dirs = statement.

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 have 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 =

let 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).

Do not confuse the input port with the output port. These are separate ports on all except the 8 pin PICAXE. The command

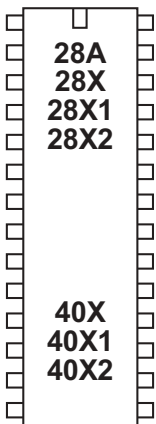
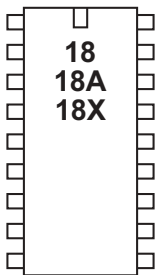
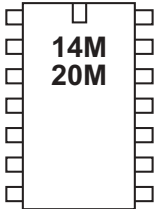
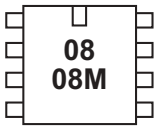
`let pins = pins`

means 'make the output port the same as the input port'.

Note that on devices that have input/output bi-directional pins (08/08M), 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 = %11000011    \ switch outputs 7,6,0,1 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).

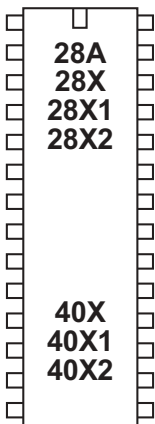
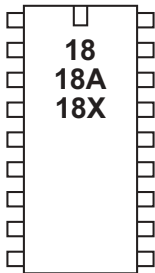
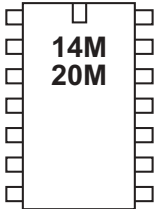
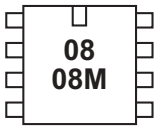
Information:

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.

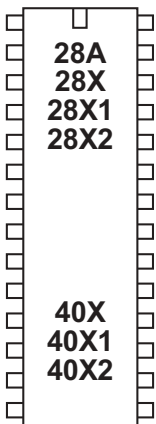
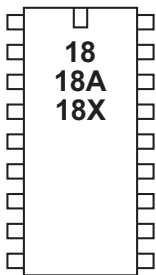
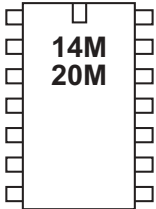
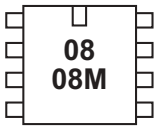
Example:

main:

```

let b0 = b0 + 1           \ increment b0
lookup b0, ("abcd"), b1  \ put ascii character into b1
if b0 < 4 then main      \ loop
end

```

**low***Syntax:***LOW** pin {,pin,pin...}

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Make pin output low.

Information:

The low command switches an output off (low).

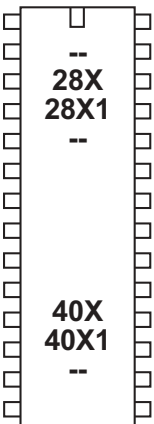
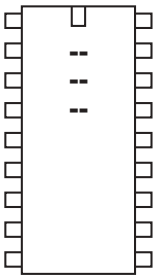
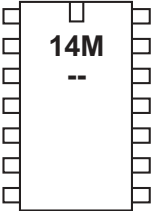
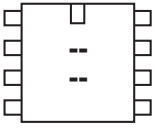
On microcontrollers with configurable input/output pins (e.g. PICAXE-08) this command also automatically configures the pin as an output.

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

```



low portc

Syntax:

LOW PORTC pin {,pin,pin...}

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

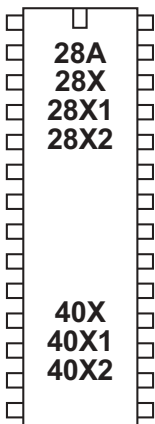
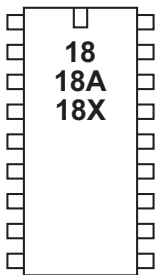
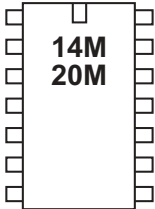
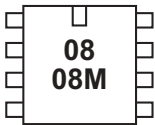
Make pin on portc output low.

Information:

The high command switches a portc output off (low).

Example:

```
main: high portc 1    \ switch on output 1
      pause 5000      \ wait 5 seconds
      low portc 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.

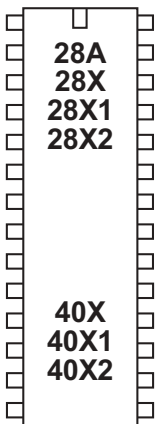
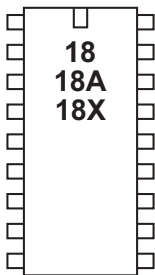
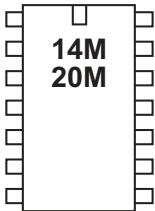
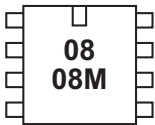
Period	Time Delay
0	18ms
1	32ms
2	72ms
3	144ms
4	288ms
5	576ms
6	1.152 s
7	2.304 s

Affect of increased clock speed:

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

Example:

```
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
```



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:

```

reset: let b1 = 0
       low 0
       low 1
       low 2
       low 3

main:  let b1 = b1 + 1
       if b1 > 3 then reset
       on b1 goto btn0,btn1, btn2, btn3
       goto main

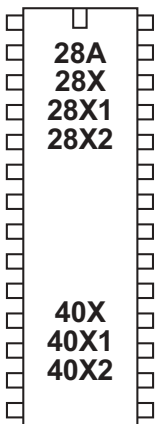
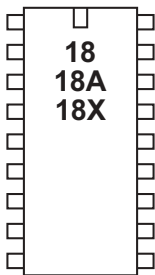
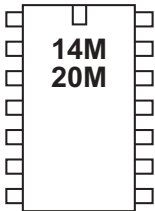
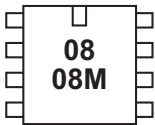
btn0:  high 0
       goto main

btn1:  high 1
       goto main

btn2:  high 2
       goto main

btn3:  high 3
       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

Example:

```

reset: let b1 = 0
       low 0
       low 1
       low 2
       low 3

main:  let b1 = b1 + 1
       if b1 > 3 then reset
       on b1 gosub btn0,btn1, btn2, btn3
       goto main

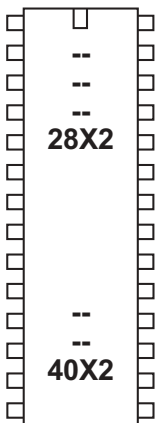
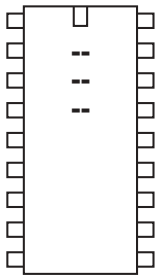
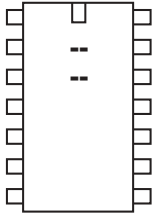
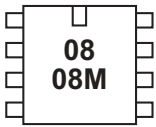
btn0:  high 0
       return

btn1:  high 1
       return

btn2:  high 2
       return

btn3:  high 3
       return

```



output

Syntax:

OUTPUT pin, pin, pin...

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Make pin an output.

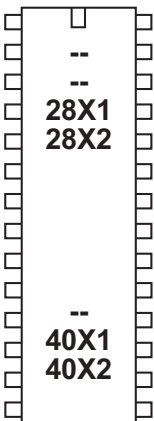
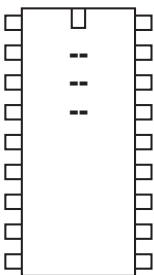
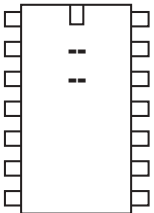
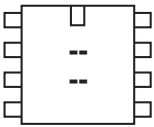
Information:

This command is only required on microcontrollers with programmable input/output pins (e.g. 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 (apart from out0 on the PICAXE-08, which is always an output).

Example:

```
main:
    input 1          \ make pin input
    reverse 1       \ make pin output
    reverse 1       \ make pin input
    output 1        \ make pin output
```

owin

Syntax:

owin pin,mode,(variable, variable...)

- Pin is a variable/constant (0-7) which specifies the input pin to use.

- Mode is a variable/ constant which selects the mode.

Each bit of 'mode' has a separate function:

bit 0 - reset pulse sent before data

bit 1 - reset pulse sent after data

bit 2 - bit mode (receive 1 bit rather than 8 bits (1 byte))

bit 3 - apply strong pullup after data

For convenience these predefined constants may be used:

0	ownoreset	4	ownoreset_bit
1	owresetbefore	5	owresetbefore_bit
2	owresetafter	6	owresetafter_bit
3	owresetboth	7	owresetboth_bit

- Variable(s) receives the data.

Function:

Read data (either full byte or single bit) from one-wire device connected to an input pin, with optional reset pulses before and after the read.

Information:

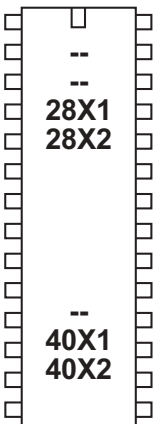
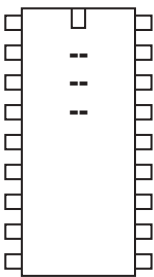
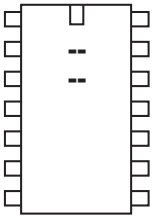
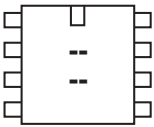
Use of one-wire parts is covered in more detail in the separate 'One-Wire Tutorial' datasheet.

This command is used to read data from a one-wire device.

Example:

```
; Read raw temperature value from DS18B20
; (this achieves a similar function to the readtemp12 command)

main:
  owout 1,%1001,($CC,$44)
                                \ send 'reset' then 'skip ROM'
                                \ then 'convert' then apply 'pullup'
  pause 750                      \ wait 750ms with strong pullup
  owout 1,%0001,($CC,$BE)
                                \ send 'reset' then 'skip ROM'
                                \ then 'read temp' command
  owin 1,%0000,(b0,b1)          \ read in result
  sertxd (#w0)                  \ transmit value
  goto main
```



owout

Syntax:

owout pin,mode,(variable,variable...)

- Pin is a variable/constant (0-7) which specifies the input pin to use.

- Mode is a variable/ constant which selects the mode.

Each bit of 'mode' has a separate function:

bit 0 - reset pulse sent before data

bit 1 - reset pulse sent after data

bit 2 - bit mode (send 1 bit rather than 8 bits (1 byte))

bit 3 - apply strong pullup after data

For convenience these predefined constants may be used:

0	ownoreset	4	ownoreset_bit
1	owresetbefore	5	owresetbefore_bit
2	owresetafter	6	owresetafter_bit
3	owresetboth	7	owresetboth_bit

- Variable(s) contain the data to be sent.

Function:

Write data to one-wire device connected to an input pin, with optional reset pulses before and after the write.

Information:

Use of one-wire parts is covered in more detail in the separate 'One-Wire Tutorial' datasheet.

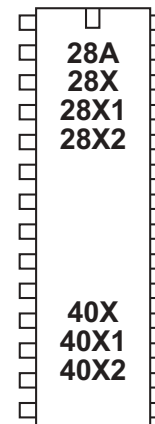
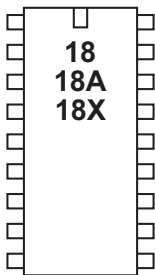
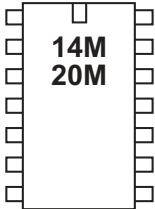
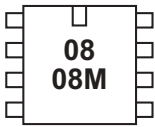
This command is used to write data to a one-wire device. Some devices, such as the DS18B20 temperature sensor, may require a strong pullup after a byte is written.

Example:

```
; Read raw temperature value from DS18B20
; (this achieves a similar function to the readtemp12 command)
```

main:

```
owout 1,%1001,($CC,$44)
                                \ send 'reset' then 'skip ROM'
                                \ then 'convert' then apply 'pullup'
pause 750                        \ wait 750ms with strong pullup
owout 1,%0001,($CC,$BE)
                                \ send 'reset' then 'skip ROM'
                                \ then 'read temp' command
owin 1,%0000,(b0,b1)           \ read in result
sertxd (#w0)                   \ transmit value
goto main
```



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 at 4MHz). The longest time delay possible is just over 65 seconds. To create a longer time delay (e.g. 5 minutes) use a for...next loop

```
for b1 = 1 to 5   ` 5 loops
  pause 60000    ` wait 60 seconds
next b1
```

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.

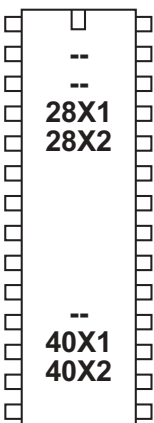
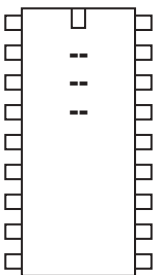
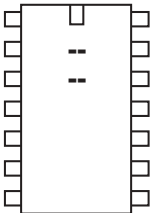
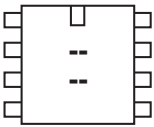
When using time delays longer than 5 seconds it may be necessary to perform a 'hard reset' to download a new program to the microcontroller. See the Serial Download section for more details.

Affect of increased clock speed:

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

Example:

```
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
```



pauseus

Syntax:

PAUSEUS *microseconds*

- *Microseconds* is a variable/constant (0-65535) which specifies how many multiples of 10 microseconds 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:

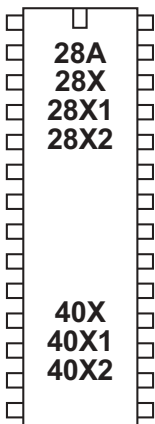
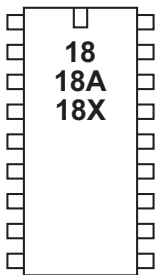
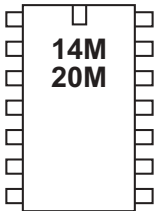
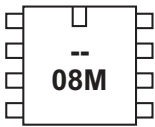
The pauseus command creates a time delay (in multiples of 10 microseconds at 4MHz). As it takes a discrete amount of time to execute the command, small time delays may be inaccurate due to this 'overhead processing' time. This inaccuracy decreases as the delay gets longer.

Affect of increased clock speed:

The timebase is reduced to 5us at 8MHz and 2.5us at 16MHz.

Example:

```
main: high 1           ` switch on output 1
      pauseus 5000     ` wait 50 000us = 50 milliseconds
      low 1            ` switch off output 1
      pauseus 5000     ` wait 50 000us = 50 milliseconds
      goto main        ` loop back to start
```



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 b0-b13.

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 to b13) to be re-used in calculations.

Addresses \$50 to \$7E are general purpose registers that can be used freely.

Addresses \$C0 to \$EF can also be used by PICAXE-18X.

Addresses \$C0 to \$FF can also be used by PICAXE-28X, 40X

Addresses \$C0 to \$EF can also be used by PICAXE-28X1, 40X1

Addresses \$40 to \$FF can also be used by PICAXE-28X2, 40X2

The second function of the peek command is for experienced users to study the internal microcontroller SFR (special function registers).

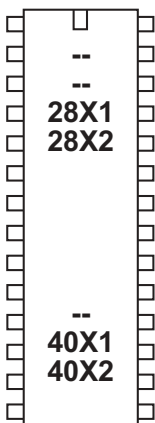
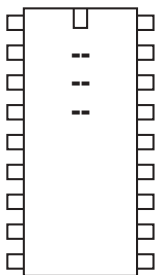
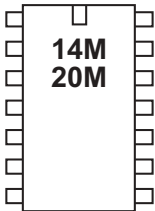
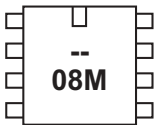
Addresses \$00 to \$1F and \$80 to \$9F are special function registers (e.g. PORTB) which determine how the microcontroller operates. Avoid using these addresses unless you know what you are doing! The command uses the microcontroller FSR register which can address register banks 0 and 1 only.

Addresses \$20 to \$4F and \$A0 to \$BF are general purpose registers reserved for use with the PICAXE bootstrap interpreter. Poking these registers will produce unexpected results and could cause the interpreter to crash.

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

Example:

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



play



Syntax:

PLAY tune,LED (PICAXE-08M only)

PLAY pin, tune (all other parts)

- pin is a variable/constant (0-7) which specifies the i/o pin to use (not available on the PICAXE-08M, which is fixed to output 2).
- Tune is a variable/constant (0 - 3) which specifies which tune to play

0 - Happy Birthday	(All parts)
1 - Jingle Bells	(08M/14M/20M only)
2 - Silent Night	(08M only)
3 - Rudolph the Red Nosed Reindeer	(08M only)
- LED is a variable/constant (0 -3) which specifies if other PICAXE-08M 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 on the PICAXE output pin.

Description:

Some PICAXE chips can play musical tones. The PICAXE is supplied with up to 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.

See the Tune command for suitable piezo / speaker circuits.

The PICAXE-08M has 4 internal tunes, other parts have less. However on these other parts the 'missing' tunes (Silent Night / Rudolph etc.) can be easily duplicated if desired via use of the tune command.

Affect of increased clock speed:

The tempo (speed) of the tune is doubled at 8MHz!

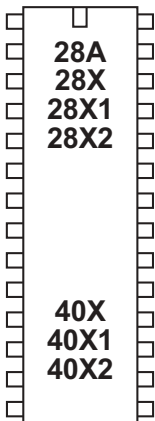
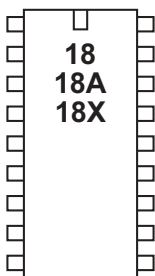
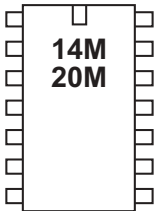
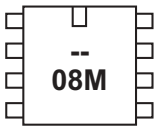
Example:

'(PICAXE-08M only)

```
play 3,1  ` rudolf red nosed reindeer with output 0 flashing
```

'(not PICAXE-08M)

```
play 2,1  ` jingle bells on output pin 2
```



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-b13.

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 to b13) 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 \$50 to \$7E are general purpose registers that can be used freely.

Addresses \$C0 to \$EF can also be used by PICAXE-18X.

Addresses \$C0 to \$FF can also be used by PICAXE-28X, 40X

Addresses \$C0 to \$EF can also be used by PICAXE-28X1, 40X1

Addresses \$40 to \$FF can also be used by PICAXE-28X2, 40X2

The second function of the poke command is for experienced users to write values to the internal microcontroller SFR (special function registers) (not 28X2).

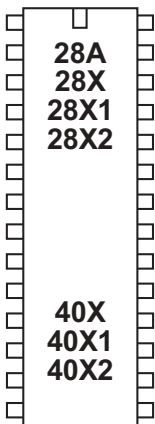
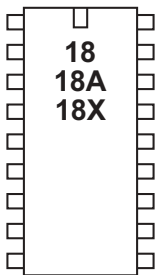
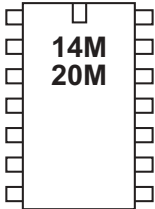
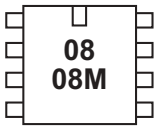
Addresses \$00 to \$1F and \$80 to \$9F are special function registers (e.g. PORTB) which determine how the microcontroller operates. Avoid using these addresses unless you know what you are doing! The command uses the microcontroller FSR register which can address register banks 0 and 1 only.

Addresses \$20 to \$4F and \$A0 to \$BF are general purpose registers reserved for use with the PICAXE bootstrap interpreter. Poking these registers will produce unexpected results and could cause the interpreter to crash.

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:

```
poke 80,b1      ` save value of b1 in register 80
poke 80, word w1
```



pulsin

Syntax:

PULSIN *pin,state,wordvariable*

- Pin is a variable/constant (0-7) 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 pulsing command measures the length of a pulse. If 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.

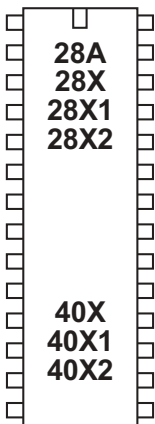
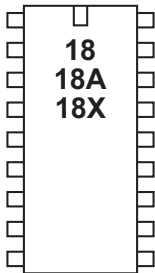
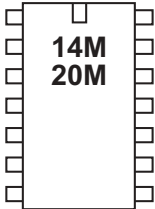
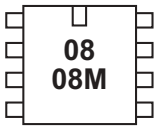
It is normal to use a word variable with this command.

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 b1
```

pulsout

Syntax:

PULSOUT pin,time

- Pin is a variable/constant (0-7) 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.

Restrictions:

The 14M only supports pulsout on outputs 0,1,2.

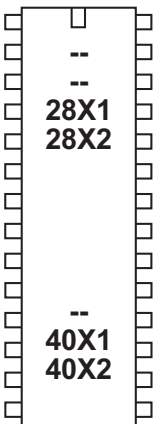
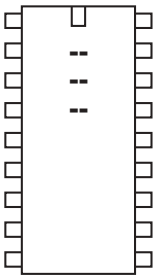
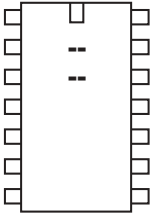
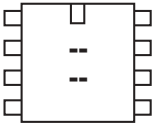
Affect of Increased Clock Speed:

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
```



put

Syntax:

PUT location,data,data,WORD wordvariable...

- Location is a variable/constant specifying a scratchpad address. Valid values are 0 to 255 (0 to 127 for X1).
- 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 scratchpad location.

Information:

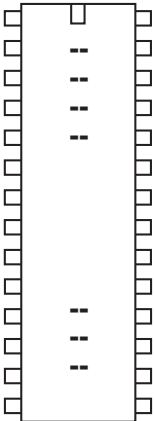
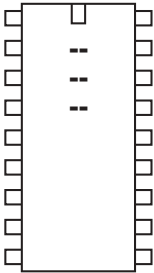
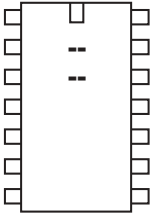
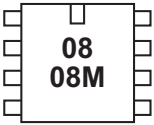
The function of the put/get commands is store temporary byte data in the microcontrollers scratchpad memory. This allows the general purpose variables (b0, b1, etc.) to be re-used in calculations.

Put and get have no affect on the scratchpad pointer and so the address next used by the indirect pointer (ptr) will not change during these commands.

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:

```
put 1,b1           ` save value of b1 in register 1
put 1, word w1
```



pwm

Syntax:

PWM pin,duty,cycles

- Pin is a variable/constant (1-4) which specifies the i/o pin to use.
- Duty is a variable/constant (0-255) which specifies analog level.
- Cycles is a variable/constant (0-255) which specifies number of cycles. Each cycle takes about 5ms.

Function:

Output pwm then return pin to input.

Information:

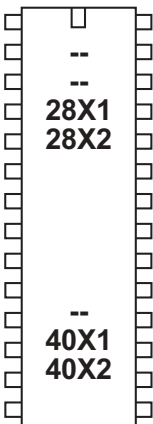
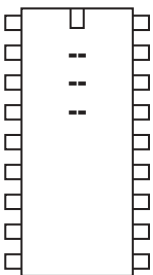
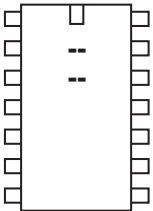
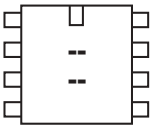
This command is rarely used. For pwm control of motors etc. the pwmout command is recommended instead.

This pwm command is used to provide 'bursts' of PWM output to generate a pseudo analogue output on the PICAXE-08/08M (pins 1, 2, 4). This is achieved with a resistor connected to a capacitor connected to ground; the resistor-capacitor junction being the analog output. PWM should be executed periodically to update/refresh the analog voltage.

Example:

main:

```
pwm 4,150,20    \ send 20 pwm bursts out of pin 4
pause 20        \ pause 20 ms
goto main       \ loop back to start
```



pwmduity

Syntax:

PWMDUTY pin,duty cycles

- Pin is a variable/constant which specifies the i/o pin to use. Note that the pwmout pin is not always a default output pin - see the pinout diagram.
 - 28X1/X2 - 1 or 2
 - 40X1/X2 - 1 or 2
- Duty is a variable/constant (0-1023) which sets the PWM duty cycle. (duty cycle is the mark or 'on time')

Function:

Alter the duty cycle after a pwmout command has been issued..

Information:

On X1 and X2 parts the pwmduity command can be used to alter the pwm duty cycle without resetting the internal timer (as occurs with a pwmout command). A pwmout command must be issued on the appropriate pin before this command will function.

Information:

See the pwmout command for more details.

Example:

init:

```
pwmout 2,150,100 ' start pwm
```

main:

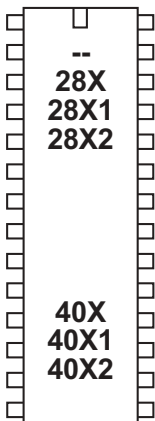
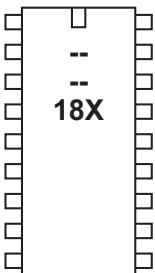
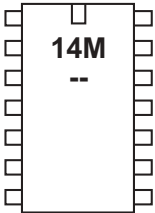
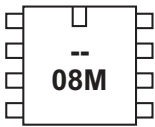
```
pwmout 2,150,150 ' set pwm duty
```

```
pause 1000 ' pause 1 s
```

```
pwmduity 2,150,50 ' set pwm duty
```

```
pause 1000 ' pause 1 s
```

```
goto main ' loop back to start
```



pwmout

Syntax:

PWMOUT pin,period,duty cycles

PWMOUT pin OFF

- Pin is a variable/constant which specifies the i/o pin to use. Note that the pwmout pin is not always a default output pin - see the pinout diagram.

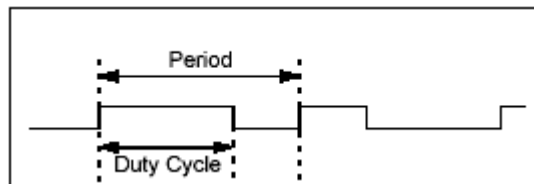
08M/14M	- 2
18X	- 3
28X/X1/X2	- 1 or 2
40X/X1/X2	- 1 or 2

- Period is a variable/constant (0-255) which sets the PWM period (period is the length of 1 on/off cycle i.e. the total mark:space time).
- Duty is a variable/constant (0-1023) which sets the PWM duty cycle. (duty cycle is the mark or 'on time')

Function:

Generate a continuous pwm output using the microcontroller's internal pwm module. For 14M and X1/X2 parts also see the HPWM command, which can produce the equivalent of pwmout on different output pins.

Information:



This command is **different** to most other BASIC commands in that the pwmout **runs continuously** (in the background) until another pwmout command is sent. Therefore it can be used, for instance, to continuously drive a motor at varying speeds. To stop pwmout send a pwmout command with period = 0.

The PWM period = (period + 1) x 4 x resonator speed

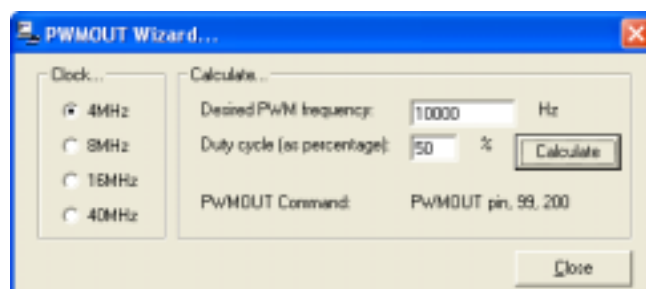
(resonator speed for 4MHz = 1/4000000)

The PWM duty cycle = (duty) x resonator speed

Note that the period and duty values are linked by the above equations. If you wish to maintain a 50:50 mark-space ratio whilst increasing the period, you must also increase the duty cycle value appropriately. A change in resonator will change the formula.

NB: If you wish to know the frequency, PWM frequency = 1 / (the PWM period)

In many cases you may want to use these equations to setup a duty cycle at a known frequency = e.g. 50% at 10 kHz. The Programming Editor software contains a wizard to automatically calculate the period and duty cycle values for you in this situation.



Select the PICAXE>Wizards>pwmout menu to use this wizard.

As the pwmout command uses the internal pwm module of the microcontroller there are certain restrictions to its use:

- 1) The command only works on certain pins.
- 2) Duty cycle is a 10 bit value (0 to 1023). The maximum duty cycle value must not be set greater than 4x the period, as the mark 'on time' would then be longer than the total PWM period (see equations above)! Setting above this value will cause erratic behaviour.
- 3) The pwmout module uses a single timer for both pins on 28/40 pin devices. Therefore when using PWMOUT on both pins the period will be the same for both pins.
- 4) The servo command cannot be used at the same time as the pwmout command as they both use the same timer.
- 5) pwmout stops during nap, sleep, or after an end command
- 6) pwmout 1 can be used at the same time as hpwm (see 3 above)
- 7) pwmout 2 cannot be used at the same time as hpwm

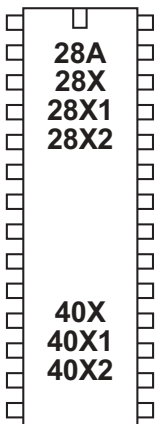
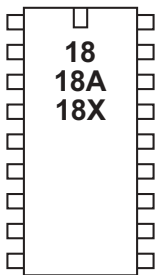
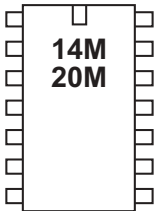
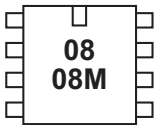
To stop pwmout on a pin it is necessary to issue a pwmout pin stop command.

The pwmout command initialises the pin for pwm operation and starts the internal timers. As each pwmout command resets the internal timer, the pwmduty command is recommended on X1 parts (after the initial pwmout) to rapidly change the duty.

When driving a FET, a pull-down resistor between the PICAXE pin and 0V is essential. The purpose of the pull-down resistor is to hold the FET driver in the correct off state whilst the PICAXE chip initialises upon power up. During this short initialisation period the pwmout pins are not actively driven (ie they 'float') and so the resistor is essential to hold the FET in the off condition.

Example:

```
main:
    pwmout 2,150,150  ` set pwm duty
    pause 1000       ` pause 1 s
    pwmduty 2,150,50 ` set pwm duty
    pause 1000       ` pause 1 s
    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 wordvariable.

Description:

The random command generates a pseudo-random sequence of numbers between 0 and 65535. All microcontrollers must perform mathematics to generate random numbers, and so 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.

When used with X1 and X2 parts you can set the timer running and then use the timer variable to 'seed' the random command. This will give much better results:

```
let w0 = timer      \ seed w0 with timer value
random w0          \ put random value into w0
```

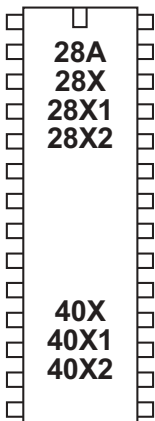
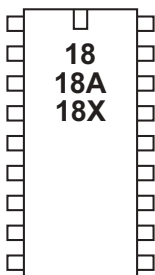
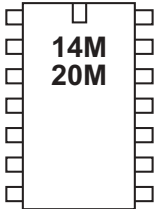
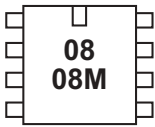
Another common way to overcome this issue (can be used on all parts) is to repeatedly call the random command within a loop, e.g. whilst waiting for a switch push. As the number of loops will vary between switch pushes, the output is much more random.

If you only require a byte variable (0-255), still use the word variable (e.g. 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 (0-7)

- 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 - check the pinout diagrams for the PICAXE chip you are using.

On microcontrollers with 'shared' inputs the ADC pin is also a digital input pin.

On microcontrollers with 'separate' inputs the ADC pins are separate pins.

	Qty	Resolution	Type	ADC Channels
PICAXE-08	1	low	shared	1
PICAXE-08M	3	10 bit	shared	1,2,4
PICAXE-14M	2(5)	10 bit	shared	0,4 (1,2,3)
PICAXE-20M	4	10 bit	shared	1,2,3,7
PICAXE-18	3	low	shared	0,1,2
PICAXE-18A	3	8 bit	shared	0,1,2
PICAXE-18X	3	10 bit	shared	0,1,2
PICAXE-28A	4	8 bit	separate	0,1,2,3
PICAXE-28X	4	10 bit	separate	0,1,2,3
PICAXE-28X1	4	10 bit	separate	0,1,2,3
PICAXE-28X2	9	10 bit	separate	0-12
PICAXE-40X	7	10 bit	separate	0,1,2,3,5,6,7
PICAXE-40X2	12	10 bit	separate	0-12

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 the supply voltage (e.g. 4.5V or 5V).

See also the calibadc command.

An 8-bit resolution analogue input will provide 256 different analogue readings (0 to 255) over the full voltage range (e.g. 0 to 5V). A low-resolution analogue input will provide 16 readings over the lower two-thirds of the voltage range (e.g. 0 to 3.3V). No readings are available in the upper third of the voltage range.

To ensure consistency between standard and low-resolution analogue input readings, the low-resolution reading on PICAXE-08 and 18 will 'jump' in 16 discrete steps between the nearest standard 8-bit readings, according to the table below.

Standard 8 Bit Reading	Low Resolution Reading
0-10	0
11-20	11
21-31	21
32-42	32
43-52	43
53-63	53
64-74	64
75-84	75
85-95	85
96-106	96
107-116	107
117-127	117
128-138	128
139-148	139
149-159	149
160-170	160
Values greater than 170 (170-255)	160

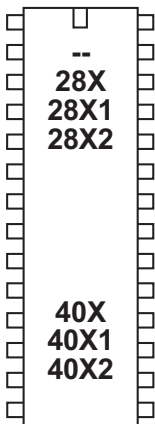
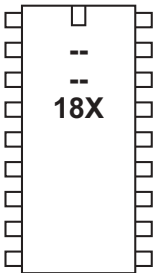
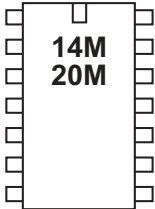
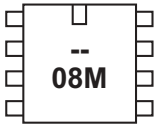
Example:

main:

```
readadc 1,b1      \ read value into b1
if b1 > 50 then flsh \ jump to flsh if b1 > 50
goto main        \ else loop back to start
```

flsh:

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



readadc10

Syntax:

READADC10 channel,wordvariable

- channel is a variable/constant specifying the input pin (0-7)

- wordvariable receives the data word read.

Function:

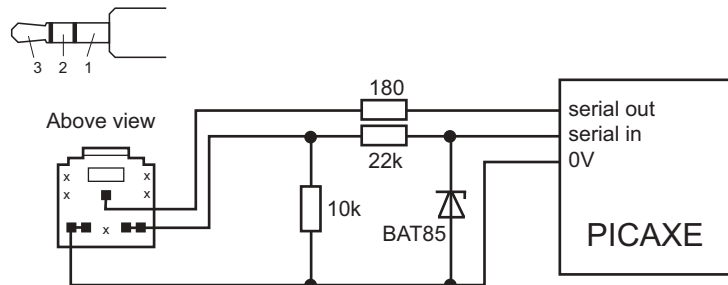
Read the ADC channel (10 bit resolution) contents into wordvariable.

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 - check the table under 'readadc' command for the PICAXE chip you are using.

As the result is 10 bit a word variable must be used - for a byte value use the readadc command instead.

When using the debug command to output 10 bit numbers, the electrical connection to the computer via the serial download cable may slightly affect the ADC values. In this case it is recommended that the 'enhanced' interface circuit is used on a serial connection (cable AXE026, not required with USB cable AXE027 as this uses 5V logic anyway). The Schottky diode within this circuit reduces this affect.



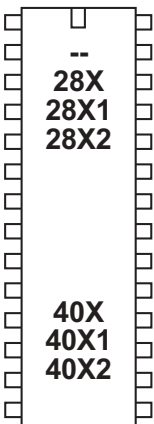
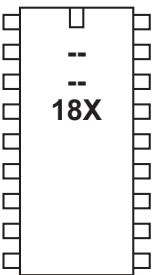
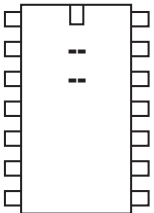
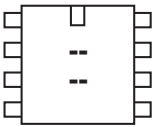
Example:

main:

```

readadc 1,w1      \ read value into b1
debug w1         \ transmit to computer
pause 200        \ short delay
goto main        \ loop back to start

```



readi2c

Syntax:

READI2C (variable,...)

READI2C location,(variable,...)

- Location is a optional variable/constant specifying a byte or word address.
- Variable(s) receives the data byte(s) read.

Function:

Read i2c location contents into variable(s).

For X1 and X2 parts the hi2cin command is recommended.

Information:

Use of i2c parts is covered in more detail in the separate 'i2c Tutorial' datasheet.

This command is used to read byte data from an i2c device. Location defines the start address of the data read, although it is also possible to read more than one byte sequentially (if the i2c device supports sequential reads).

Location must be a byte or word as defined within the i2cslave command. An i2cslave command must have been issued before this command is used.

If the i2c hardware is incorrectly configured, or the wrong i2cslave data has been used, the value 255 (\$FF) will be loaded into each variable.

Example:

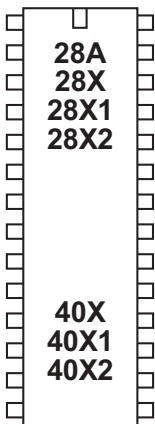
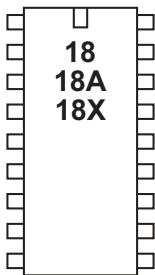
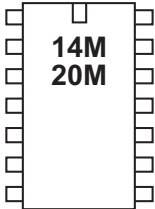
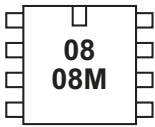
```
; Example of how to use DS1307 Time Clock
; Note the data is sent/received in BCD format.

' set DS1307 slave address
  i2cslave %11010000, i2cslow, i2cbyte

' read time and date and debug display
```

main:

```
readi2c 0,(b0,b1,b2,b3,b4,b5,b6,b7)
debug b1
pause 2000
goto main
```



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 (e.g. for w0, read both b0 and b1).

With the PICAXE-08, 08M, 14M and 18 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-08, 18	0 to (127 - number of used bytes)
PICAXE-08M, 14M, 20M	0 to (255 - number of used bytes)

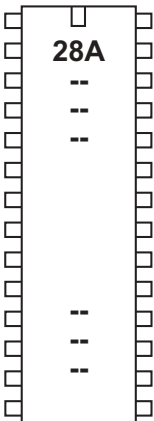
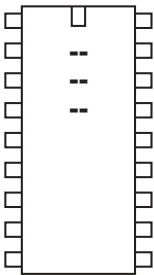
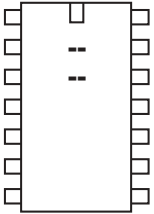
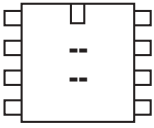
With the following microcontrollers the data memory is completely separate from the program and so no conflicts arise. The number of bytes available varies depending on microcontroller type as follows.

PICAXE-28, 28A	0 to 63
PICAXE-28X, 40X	0 to 127
PICAXE-18A, 18X, 28X1, 28X2, 40X1, 40X2	0 to 255

When word variables are used (with the keyword **WORD**) the two byte sof 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 into b1
    serout 7,T2400,(b1)    ` transmit value to serial LCD
  next b0                  ` next loop
```



readmem

Syntax:

READMEM location,data

- Location is a variable/constant specifying a byte-wise address (0-255).
- Data is a variable into which the data is read.

Function:

Read FLASH program memory byte data into variable.

Information:

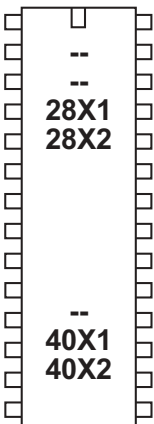
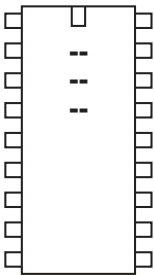
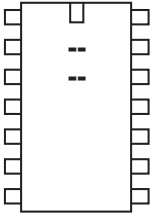
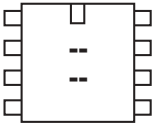
The data memory on the PICAXE-28A is limited to only 64 bytes. Therefore the readmem command provides an additional 256 bytes storage in a second data memory area. This second data area is not reset during a download.

This command is not available on the PICAXE-28X as a larger i2c external EEPROM can be used.

The readmem 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 (e.g. for w0, read both b0 and b1).

Example:

```
main: for b0 = 0 to 255           \ start a loop
      readmem b0,b1             \ read value into b1
      serout 7,T2400,(b1)       \ transmit value to serial LCD
next b0                          \ next loop
```



readtable

Syntax:

`readtable location,variable`

- location is a variable/constant specifying a byte-wise address (0-255).
- variable receives the current value of the scratchpad pointer

Function:

Read the value from an embedded lookup table.

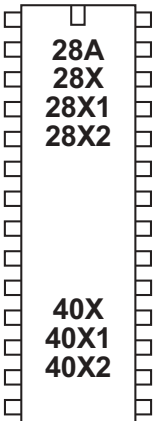
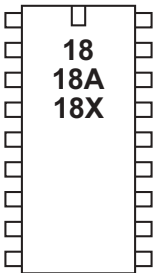
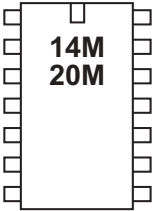
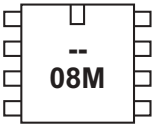
Information:

Some PICAXE chips enable lookup data (e.g. LCD messages) to be embedded in a table within the program (via the table command). This is a very efficient way of storing data.

Example:

```
TABLE 0,("Hello World")      ` save values in table

main:
  for b0 = 0 to 10            ` start a loop
    readtable b0,b1          ` read value from table
    serout 7,N2400,(b1)      ` transmit to serial LCD module
  next b0                     ` next character
```



readoutputs

Syntax:

READOUTPUTS *variable*

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

Function:

Read the output pins value into variable.

Information:

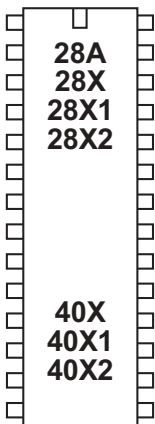
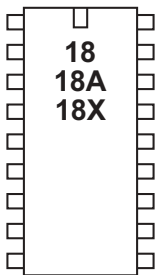
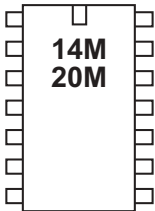
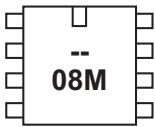
The current state of the output pins can be read into a variable using the readoutputs command. Note that this is not the same as 'let var = pins', as this let command reads the status of the input (not output) pins.

This command is not normally used with X1 and X2 parts as the outputs can be read directly with 'let var = outpins'

Example:

main:

```
readoutputs b1          ' read outputs value into b1
```



readtemp

Syntax:

READTEMP pin,variable

- Pin is the input pin.

- Variable receives the data byte read.

Function:

Read temperature from a DS18B20 digital temperature sensor and store in variable. The conversion takes up to 750ms.

Information:

The temperature is read back in whole degree steps, and the sensor operates from -55 to + 125 degrees celsius. Note that bit 7 is 0 for positive temperature values and 1 for negative values (ie negative values will appear as 128 + numeric value).

Note the readtemp command does not work with the older DS1820 or DS18S20 as they have a different internal resolution. This command cannot be used on pin0 or pin3 of the PICAXE-08M/14M, or pin6 of the 20M.

Affect of increased clock speed:

This command only functions at 4MHz. X1 and X2 parts automatically use the internal 4MHz resonator for this command.

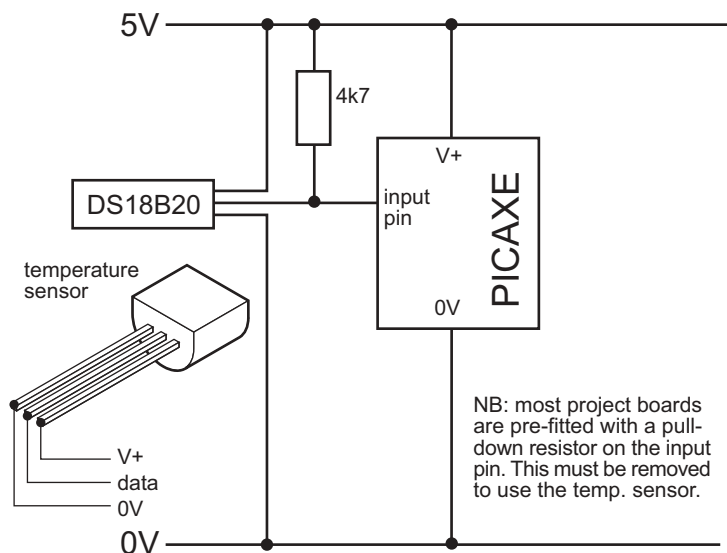
Example:

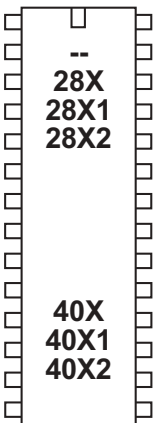
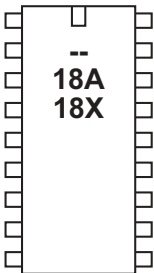
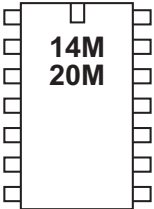
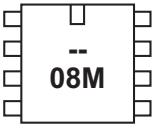
main:

```
readtemp 1,b1          \ read value into b1
if b1 > 127 then neg   \ test for negative
serout 7,N2400,(#b1)  \ transmit value to serial LCD
goto loop
```

neg:

```
let b1 = b1 - 128 \ adjust neg value
serout 7,N2400,("-") \ transmit negative symbol
serout 7,N2400,(#b1) \ transmit value to serial LCD
goto main
```





readtemp12

Syntax:

READTEMP12 pin,wordvariable

- Pin is the input pin.

- Variable receives the raw 12 bit data read.

Function:

Read 12 bit temperature data from a DS18B20 digital temperature sensor and store in variable. The conversion takes up to 750ms.

Information:

This command is for advanced users only. For standard 'whole degree' data use the readtemp command.

The temperature is read back as the raw 12 bit data into a word variable (0.125 degree resolution). The user must interpret the data through mathematical manipulation. See the DS18B20 datasheet (www.dalsemi.com) for more information on the 12 bit Temperature/Data relationship.

See the readtemp command for a suitable circuit.

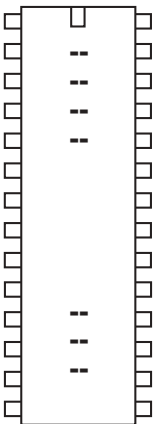
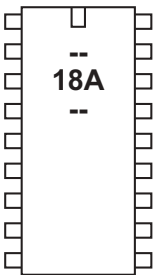
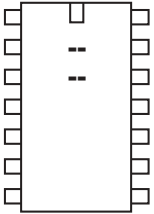
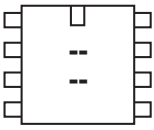
Note the readtemp12 command does not work with the older DS1820 or DS18S20 as they have a different internal resolution.

Affect of increased clock speed:

This command only functions at 4MHz.X1 and X2 parts automatically use the internal 4MHz resonator for this command.

Example:

```
main:
    readtemp12 1,w1      \ read value into b1
    debug w1            \ transmit to computer screen
    goto main
```



readowclk

Syntax:

`readowclk pin`

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Read seconds from a DS2415 clock chip.

Information:

This command only applies to the PICAXE-18A. It is now rarely used as most users prefer to use the more powerful DS1307 i2c part interfaced to a PICAXE-18X microcontroller.

The DS2415 is an accurate 'second counter'. Every second, the 32 bit (4 byte) counter is incremented. Time is very accurate due to the use of a watch crystal. Therefore by counting elapsed seconds you can work out the accurate elapsed time. The 32 bit register is enough to hold 136 years worth of seconds. If desired the DS2415 can be powered by a separate 3V cell and so continue working when the main PICAXE power is removed.

Note that after first powering the DS2415 you must use a `resetowclk` command to activate the clock crystal and reset the counter. See the circuit diagram under the `resetowclk` command description.

The `readowclk` command reads the 32 bit counter and then puts the 32 bit value in variables b10 (LSB) to b13 (MSB) (also known as w6 and w7).

Using byte variables:

The number in b10 is the number of single seconds

The number in b11 is the number x 256 seconds

The number in b12 is the number x 65536 seconds

The number in b13 is the number x 16777216 seconds

Using word variables:

The number in w6 is the number of single seconds

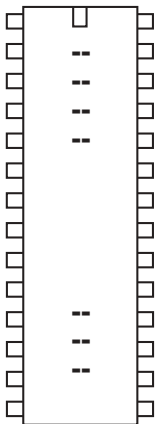
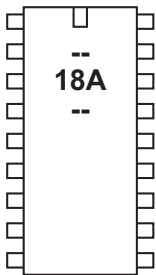
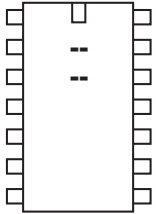
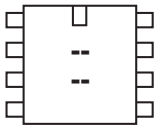
The number in w7 is the number x 65536 seconds

Affect of Increased Clock Speed:

This command will only function at 4MHz.

Example:

```
main:
    resetowclk 2      ' reset the clock on pin2
loop1:
    readowclk 2      ' read clock on input2
    debug b1         ' display the elapsed time
    pause 10000     ' wait 10 seconds
    goto loop1
```



resetowclk

Syntax:

resetowclk pin

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Reset seconds count to 0 on a DS2415 clock chip.

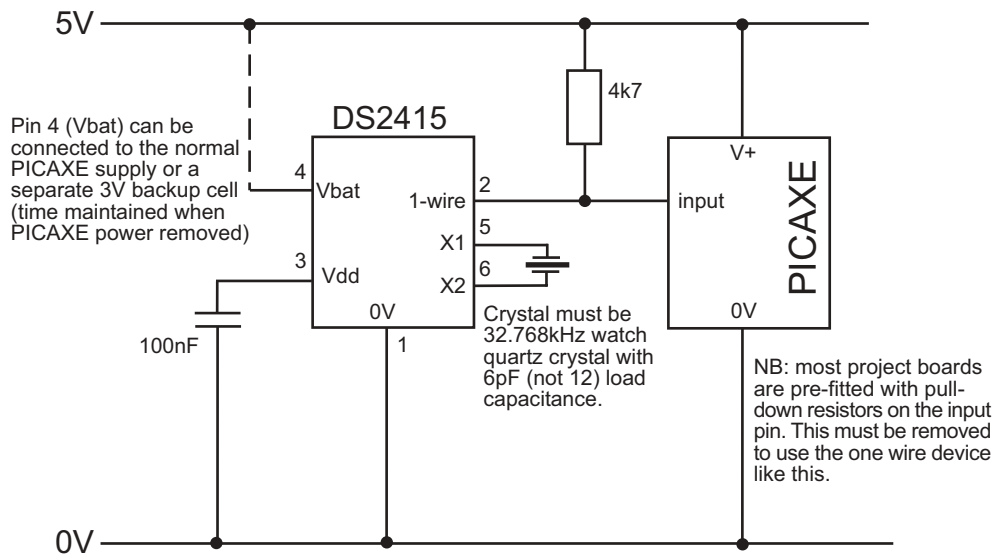
Information:

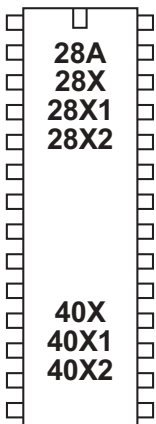
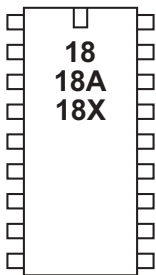
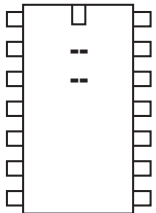
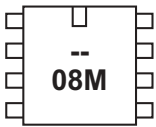
This command resets the time on a DS2415 one wire clock chip. It also switches the clock crystal on, and so must be used when the chip is first powered up to enable the time counting.

Affect of Increased Clock Speed:

This command will only function at 4MHz.

See the example under the readowclk command.





readownsn

Syntax:

`readownsn pin`

- Pin is a variable/constant (0-7) which specifies the input pin to use.

Function:

Read serial number from any Dallas 1-wire device.

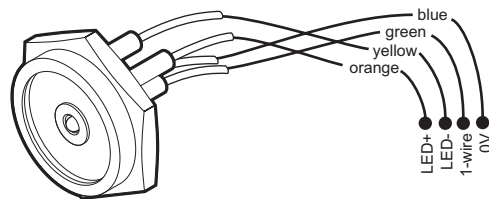
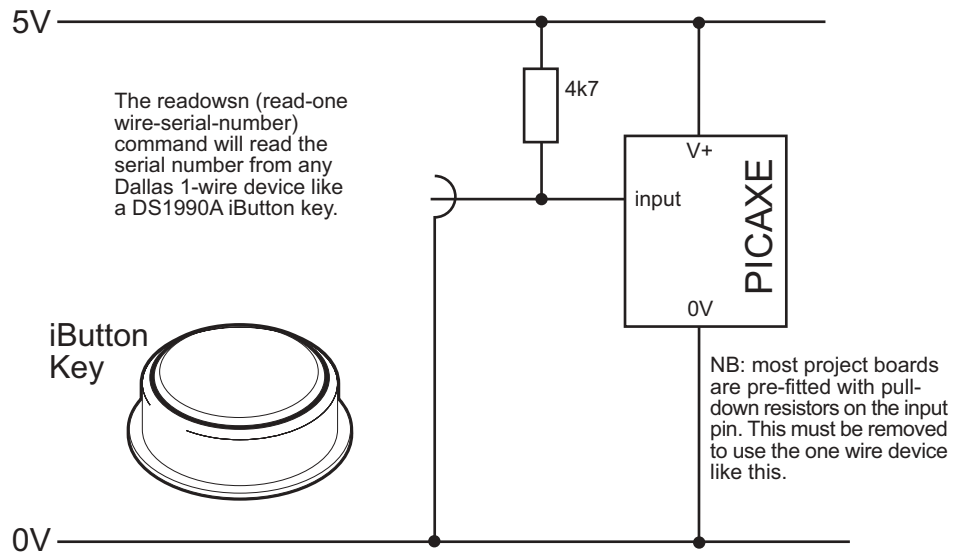
Information:

This command (read-one-wire-serial-number) reads the unique serial number from any Dallas 1-wire device (e.g DS18B20 digital temp sensor, DS2415 clock, or DS1990A iButton).

If using an iButton device (e.g. DS1990A) this serial number is laser engraved on the casing of the iButton.

The readownsn command reads the serial number and then puts the family code in b6, the serial number in b7 to b12, and the checksum in b13

Note that you should not use variables b6 to b13 for other purposes in your program during a readownsn command.



Part RSA002 - iButton Contact probe

Example:

main:

```
let b6 = 0 ' reset family code to 0
```

```
' loop here reading numbers until the  
' family code (b6) is no longer 0
```

loop1:

```
readown 2 ' read serial number on input2  
if b6 = 0 then loop1
```

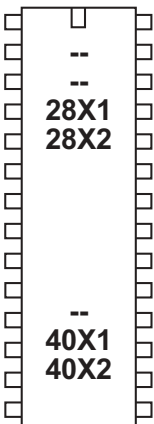
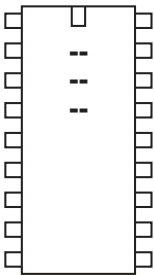
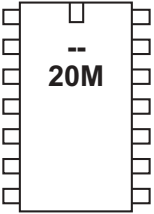
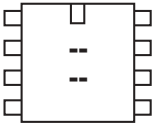
```
' Do a simple safety check here.  
' b12 serial no value will not likely be FF  
' if this value is FF, it means that the device  
' was removed before a full read was completed  
' or a short circuit occurred
```

```
if b12 = $FF then main
```

```
'Everything is ok so continue
```

```
debug b1 ' ok so display  
pause 1000 ' short delay
```

```
goto main
```



reconnect

Syntax:

RECONNECT

Function:

Reconnect a disconnected PICAXE so that it scans for new downloads.

Information:

The PICAXE chips constantly scan the serial download pin to see if a computer is trying to initialise a new program download. However when it is desired to use the download pin for user serial communication (serrxd command), it is necessary to disable this scanning.

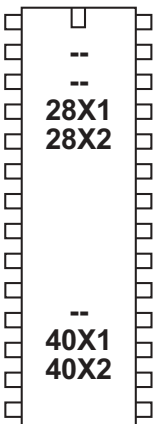
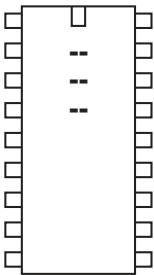
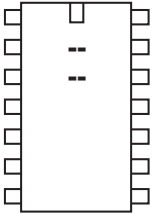
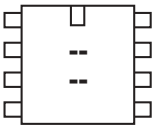
After disconnect is used it will not be possible to download a new program until:

- 1) the reconnect command is issued
- 2) a reset command is issued
- 3) a hardware reset is carried out

Remember that is always possible to carry out a new download by carrying out the 'hard-reset' procedure.

Example:

```
disconnect
serrxd [1000, timeout],@ptrinc,@ptrinc,@ptr
reconnect
```



reset

Syntax:

`reset`

Function:

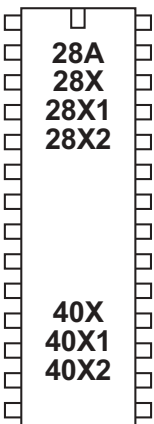
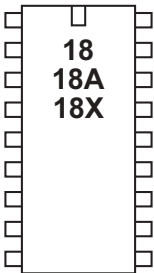
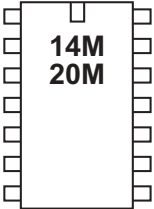
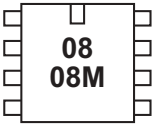
Force a chip reset. This is the software equivalent of pressing the external reset switch.

Information:

The reset command is the software equivalent of pressing the external reset switch. The program is reset to the first line and all variables, stacks etc are reset.

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
    reset           \ start again
```



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

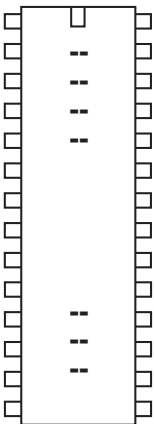
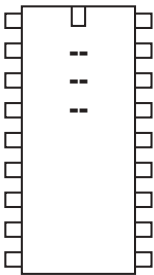
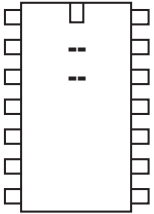
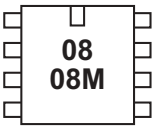
```

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 (0-7) which specifies the i/o pin to use.

Function:

Make pin an output if now input and vice versa.

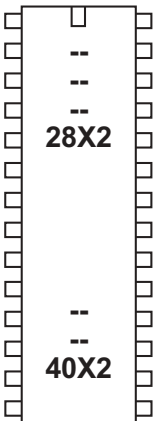
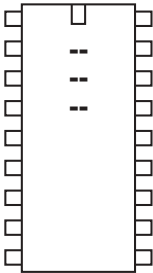
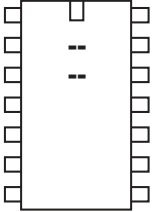
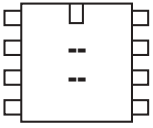
Information:

This command is only required on microcontrollers with programmable input/output pins (e.g. 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 (apart from out0 on the PICAXE-08, which is always an output). Note that pin3 is always an input.

Example:

```
main:
    input 1           \ make pin input
    reverse 1        \ make pin output
    reverse 1        \ make pin input
    output 1         \ make pin output
```



run

Syntax:

RUN slot

- slot is a variable/constant (1 or 2) which specifies which program to run

Function:

Run another program slot.

Information:

The X2 parts have two completely separate program slots. By default program 1 runs whenever the part is reset.

To run the second program use the command 'run 2'. This command stops the current program and starts the second program running immediately. Variables are not reset, so can be shared between the programs.

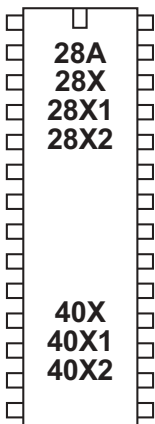
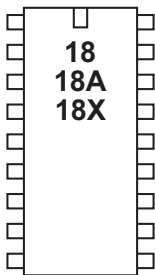
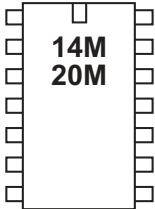
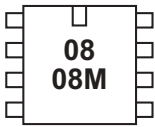
When in program 2 you can also use 'run 1' to restart the first program. If you wish to also reset the variables you must use a 'reset' command instead to restart program 1. This is equivalent to 'run 1' + variable reset.

Note that when carrying out a new program download the download is into the first program slot by default. If you wish to download into the second program slot you must use the '#slot 2' directive within the program.

Example:

```
#slot 1
init:
    if pin1 =1 then main    ` test an input upon reset
    run 2                  ` input is low so run second program

main: high 1              ` this is normal program (slot 1)
    etc.
```



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
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 a variable value and conditionally execute sections of code.

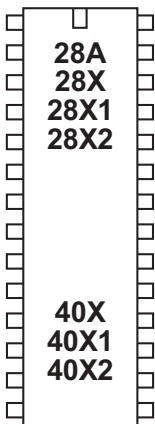
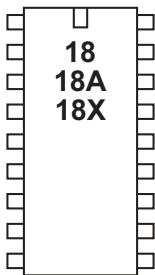
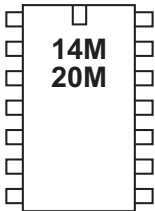
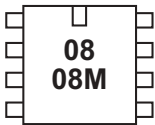
Information:

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 flow 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
case 1
    high 1
case 2,3
    low 1
case 4 to 6
    high 2
else
    low 2
endselect
```



serin

Syntax:

SERIN pin,baudmode,(qualifier,qualifier...)

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

SERIN pin,baudmode,{#}variable,{#}variable...

Additional optional timeout syntax options for X1 and X2 parts:

SERIN [timeout,address], pin,baudmode,(qualifier...)

SERIN [timeout,address], pin,baudmode,(qualifier...),{#}variable,{#}variable

SERIN [timeout,address], pin,baudmode,{#}variable,{#}variable

- Pin is a variable/constant (0-7) 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)

4MHz	8MHz	16MHz
T300_4	T600_8	T1200_16
T600_4	T1200_8	T2400_16
T1200_4	T2400_8	T4800_16
T2400_4	T4800_8	T9600_16
T4800_4	T9600_8	T19200_16
N300_4	N600_8	N1200_16
N600_4	N1200_8	N2400_16
N1200_4	N2400_8	N4800_16
N2400_4	N4800_8	N9600_16
N4800_4	N9600_8	N19200_16

(not all speeds are available on all parts)

- 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.
- Timeout is an optional variables/constants which sets the timeout period in milliseconds
- Address is a label which specifies where to go if a timeout occurs.

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. 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).

Note that the 4800 baud rate is only available on the X parts. Note that the microcontroller may not be able to keep up with processing complicated datagrams at higher speeds - in this case it is recommended that the transmitting device leaves a short delay (e.g. 2ms) between each byte.

Qualifiers are used to specify a 'marker' byte or sequence. The command

```
serin 1,N2400,("ABC"),b1
```

requires to receive the string "ABC" before the next byte read is put into byte b1

Without qualifiers

```
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")
```

The X1 and X2 parts can take an optional timeout value and address at the start of the command. The timeout value, set in milliseconds, is the length of time the serin command will wait for a serial command to be detected. After the timeout period, if no signal is detected, program flow will jump to the time out address.

After using this command you may have 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:

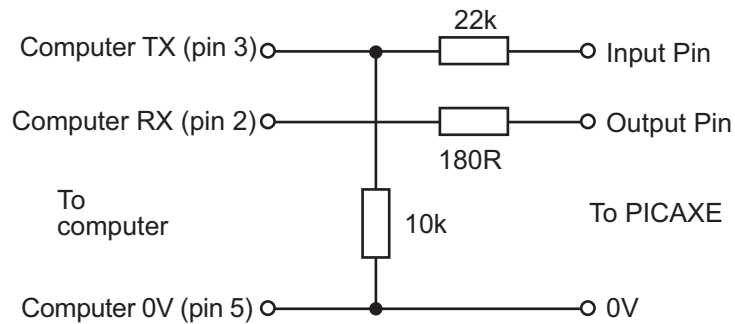
Increasing the clock speed increases the serial baud rate as shown below.

Baudmode	4MHz	8MHz	16MHz
300	300	600	1200
600	600	1200	2400
1200	1200	2400	4800
2400	2400	4800	9600
4800	4800	9600	19200

A maximum of 4800 is recommended for complicated serial transactions.

Internal resonators are not as accurate as external resonators, so in high accuracy applications an external resonator device is recommended. However microcontrollers with an internal resonator may be used successfully in most applications, and may also be calibrated using the calibfreq command if required.

Example Computer Interface Circuit:



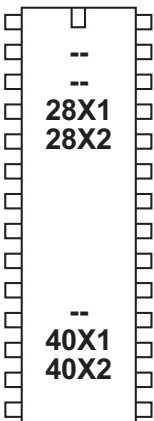
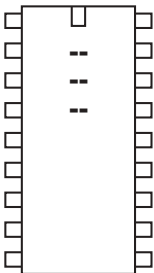
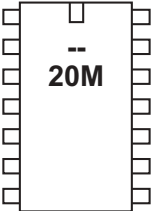
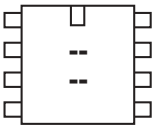
PICAXE-08/08M/14M - Due to the internal structure of input3 on these chips, a 1N4148 diode is required between the pin and V+ for serin to work on this particular pin ('bar' end of diode to V+) with this circuit. All other pins have an internal diode.

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

```



serrxd

Syntax:

SERRXD (qualifier,qualifier...)

SERRXD (qualifier,qualifier...),{#}variable,{#}variable...

SERRXD {#}variable,{#}variable...

SERRXD [timeout,address], (qualifier...)

SERRXD [timeout,address], (qualifier...),{#}variable,{#}variable

SERRXD [timeout,address], {#}variable,{#}variable

- 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.
- Timeout is an optional variables/constants which sets the timeout period in milliseconds
- Address is a label which specifies where to go if a timeout occurs.

Function:

Serial input via the serial input programming pin (at fixed baud rate 4800, 8 data, no parity, 1 stop). The 20M does not support timeout options.

Information:

The serrxd command is similar to the serin command, but acts via the serial input pin rather than a general input pin. This allows data to be received from the computer via the programming cable.

The PICAXE chip normally constantly scans the serial download pin to see if a computer is trying to initialise a new program download. However when it is desired to use serrxd it is necessary to disable this scanning. This is automatic, and is effectively the same as issuing a 'disconnect' command.

After disconnect is used it will not be possible to download a new program until:

- 1) the reconnect command is issued
- 2) a reset command is issued
- 3) a hardware reset is carried out

Remember that it is always possible to carry out a new download by carrying out the 'hard-reset' procedure (described in the PICAXE manual part 1).

Affect of Increased Clock Speed:

Increasing the clock speed increases the serial baud rate as shown below.

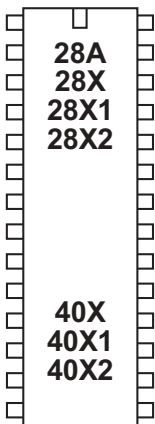
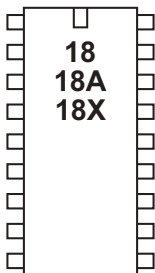
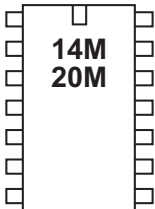
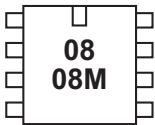
4MHz	8MHz	16MHz	32MHz
4800	9600	19200	38400

Example:

```
disconnect
```

```
serrxd [1000, timeout],@ptrinc,@ptrinc,@ptr
```

```
reconnect
```



serout



Syntax:

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

- Pin is a variable/constant (0-7) 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)

4MHz	8MHz	16MHz
T300_4	T600_8	T1200_16
T600_4	T1200_8	T2400_16
T1200_4	T2400_8	T4800_16
T2400_4	T4800_8	T9600_16
T4800_4	T9600_8	T19200_16
N300_4	N600_8	N1200_16
N600_4	N1200_8	N2400_16
N1200_4	N2400_8	N4800_16
N2400_4	N4800_8	N9600_16
N4800_4	N9600_8	N19200_16

(not all speeds are available on all parts)

- 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 sertxd 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).

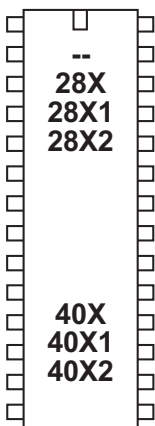
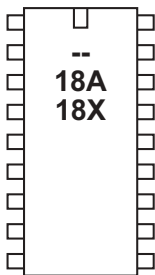
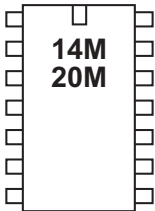
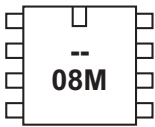
Note that the 4800 baud rate is only available on the X parts.

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.

Please also see the interfacing circuits, affect of resonator clock speed, and explanation notes of the 'serin' command, as all of these notes also apply to the serout command.

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
```



sertxd

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

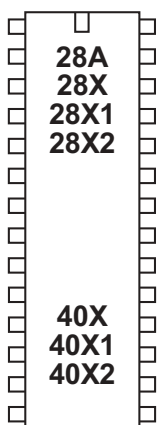
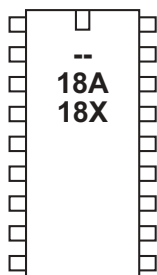
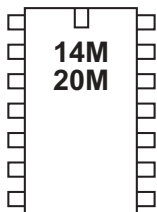
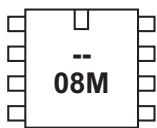
Affect of Increased Clock Speed:

Increasing the clock speed increases the serial baud rate as shown below.

4MHz	8MHz	16MHz
4800	9600	19200

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
```



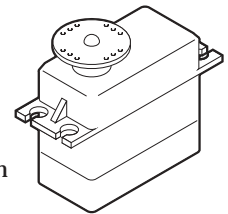
servo

Syntax:

SERVO pin,pulse

SERVO OFF

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.
- Pulse is variable/constant (75-225) which specifies the servo position



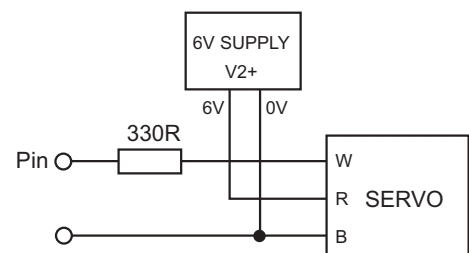
Function:

Pulse an output pin continuously to drive a radio-control style servo

Information:

Servos, as commonly found in radio control toys, are a very accurate motor/gearbox assembly that can be repeatedly moved to the same position due to their internal position sensor. Generally servos require a pulse of 0.75 to 2.25ms every 20ms, and this pulse must be constantly repeated every 20ms. Once the pulse is lost the servo will lose its position. The servo command starts a pin pulsing high for length of time pulse (x0.01 ms) every 20ms. This command is **different** to most other BASIC commands in that the pulsing mode **continues** until another servo, high or low command is executed. High and low commands stop the pulsing immediately. Servo commands adjust the pulse length to the new pulse value, hence moving the servo. Servo cannot be used at the same time as timer or pwmout/hpwm as they share a common internal timer resource.

The 'servo' command initialises the pin for servo operation and starts the timer. Once a pin has been initialised, it is recommended to use the 'servopos' command to adjust position. This prevents resetting of the timer, which could cause 'jitter'



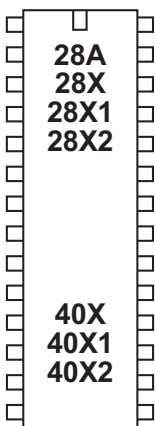
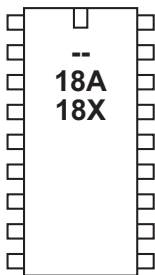
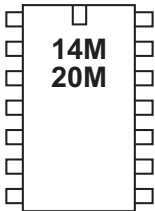
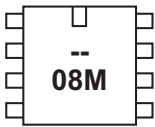
Do not use a pulse value less than 75 or greater than 225, as this may cause the servo to malfunction. Due to tolerances in servo manufacture all values are approximate and will require fine-tuning by experimentation. Always use a separate 6V (e.g 4x AA cells) power supply for the servo, as they generate a lot of electrical noise. Note that the overhead processing time required for processing the servo commands every 20ms causes the other commands to be slightly extended i.e. a pause command will take slightly longer than expected. The servo pulses are also temporarily disabled during timing sensitive serin, serout, srtxd and debug commands.

Affect of increased clock speed:

The servo command will function correctly at 4MHz on all parts. 16MHz is also supported on X1 and X2 parts. No other frequency will work correctly.

Example:

```
init: servo 4,75           \ initialise servo
main: servopos 4,75       \ move servo to one end
      pause 2000          \ wait 2 seconds
      servopos 4,225      \ move servo to other end
      pause 2000          \ wait 2 seconds
      goto main           \ loop back to start
```



servopos

Syntax:

SERVOPOS pin,pulse

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.
- Pulse is variable/constant (75-225) which specifies the servo position

Function:

Adjust the pulse length applied to a radio-control style servo to change its position. A servo command on the same pin number must have been previously issued.

Information:

Servos, as commonly found in radio control toys, are a very accurate motor/gearbox assembly that can be repeatedly moved to the same position due to their internal position sensor. Generally servos require a pulse of 0.75 to 2.25ms every 20ms, and this pulse must be constantly repeated every 20ms. Once the pulse is lost the servo will lose its position. The 'servo' command starts a pin pulsing high for length of time pulse (x0.01 ms) every 20ms. The 'servopos' adjusts the length of this pulse.

The 'servo' command initialises the pin for servo operation and starts the timer. Once a pin has been initialised, it is recommended to use the 'servopos' command to adjust position. This prevents resetting of the timer, which could cause 'jitter'

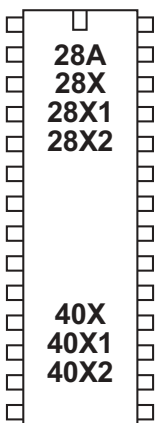
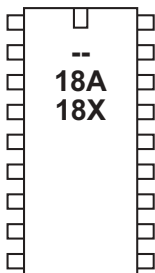
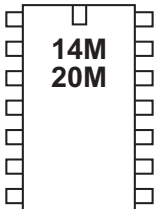
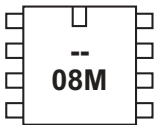
Do not use a pulse value less than 75 or greater than 225, as this may cause the servo to malfunction. Due to tolerances in servo manufacture all values are approximate and will require fine-tuning by experimentation. Always use a separate 6V (e.g 4x AA cells) power supply for the servo, as they generate a lot of electrical noise. Note that the overhead processing time required for processing the servo commands every 20ms causes the other commands to be slightly extended i.e. a pause command will take slightly longer than expected. The servo pulses are also temporarily disabled during timing sensitive serin, serout, sertxd and debug commands.

Affect of increased clock speed:

The servo command will function correctly at 4MHz on all parts. 16MHz is also supported on X1 and X2 parts. No other frequency will work correctly.

Example:

```
init: servo 4,75           \ initialise servo
main: servopos 4,75       \ move servo to one end
      pause 2000          \ wait 2 seconds
      servopos 4,225     \ move servo to other end
      pause 2000          \ wait 2 seconds
      goto main           \ loop back to start
```



setint



Syntax:

SETINT OFF

SETINT input,mask

SETINT NOT input,mask *(X1, X2 parts only)*

- 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.

X1 and X2 parts can also alternately interrupt on a certain 'flags' byte condition - see setintflags command.

Information:

The setint command causes a polled interrupt on a certain input pin / flags 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.

e.g.

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.

The X1 and X2 parts also support the NOT condition, where the interrupt occurs when the pattern is NOT as the port/mask define.. They can also use the 'flags' byte (instead of the input port) to generate the interrupt condition.

Special note - 14M/20M only

Due to the internal port configuration of the 14M only inputs 0,1,2 may be used for an interrupt condition. Inputs 3 and 4 are ignored during setint operation.

Due to the internal port configuration of the 20M only inputs 1-5 may be used for an interrupt condition. Inputs 0, 6 and 7 are ignored during setint operation.

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
- %01000000 will check input pin 6
- %10000000 will check input pin 7
- etc

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

- %00000011 will check input pins 1 and 0
- %10000100 will check input pins 7 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 %10000100 will check pins 7 and 2 and create a value of %a0000b00 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 %10000100, pins 7 and 2, then the valid 'compare with value' can be one of the following ...

- %00000000 Pin 7 = 0 and pin 2 = 0
- %00000100 Pin 7 = 0 and pin 2 = 1
- %10000000 Pin 7 = 1 and pin 2 = 0
- %10000100 Pin 7 = 1 and pin 2 = 1

So, if you want to generate an interrupt whenever Pin 7 is high and Pin 2 is low, the 'input mask' is %10000100 and the 'compare with value' is %10000000, giving a SETINT command of ...

```
- SETINT %10000000,%10000100
```

The interrupt will then occur when, and only when, pin 7 is high and pin 2 is low. If pin 7 is low or pin 2 is high the interrupt will not happen as two pins are 'looked at' in the mask.

Example:

```
setint %10000000,%10000000
` activate interrupt when pin7 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
```

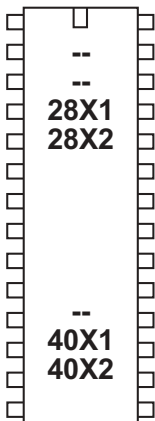
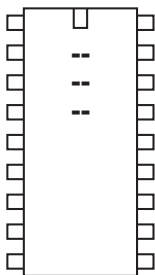
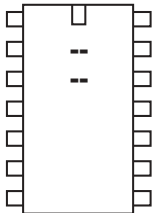
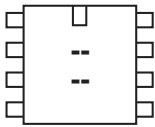
In this example an LED on output 1 will light immediately the input is switched high. With a standard if pin7 = 1 then.... type statement the program could take up to two seconds to light the LED as the if statement is not processed during the pause 2000 delay time in the main program loop (standard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



setintflags

Syntax:

SETINTFLAGS OFF

SETINTFLAGS flags,mask

SETINTFLAGS NOT flags,mask

- flags is a variable/constant (0-255) which specifies flags byte condition.

- mask is variable/constant (0-255) which specifies the mask

Function:

Interrupt on a certain 'flags' byte condition.

Information:

The setintflags command causes a polled interrupt on a certain flags condition.

A polled interrupt is a quicker way of reacting to a particular event. It is the only type of interrupt available in the PICAXE system. The flags byte 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 flags byte masked by the byte 'mask'. Therefore any bits masked by a '0' in byte mask will be ignored.

The system 'flags' byte is broken down into individual bit variables.

Name	Special	Special function
flag0	-	reserved for future use
flag1	-	reserved for future use
flag2	-	reserved for future use
flag3	-	reserved for future use
flag4	-	reserved for future use
flag5	hserflag	hserial background receive has occurred
flag6	hi2cflag	hi2c write has occurred (slave mode)
flag7	toflag	timer overflow flag

to interrupt on timer 0 overflow

```
setintflags %10000000,%10000000
```

to interrupt on hi2c write (slave mode)

```
setintflags %01000000,%01000000
```

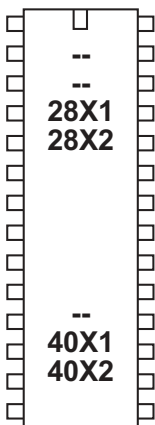
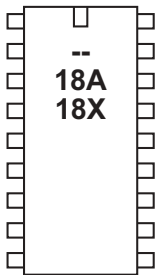
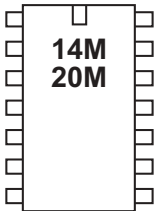
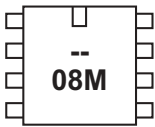
to interrupt on background hardware serial receive

```
setintflags %00100000,%00100000
```

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

Example:

```
setintflags %10000000,%10000000 `set timer 0 to interrupt
```

setfreq

Syntax:

`setfreq freq`

- freq is the keyword that selects the appropriate frequency

08M, 14M, 20M	internal	m4, m8
18A, 18X	internal	m4, m8
28X1, 40X1	internal	k31,k125,k250,k500,m1, m2, m4, m8
	external	em4, em8, em10, em16, em20
28X2, 40X2	internal	k31,k125, k250, k500,m1, m2, m4, m8
	external	em16, em32,e m40

Function:

Set the internal clock frequency for microcontrollers with internal resonator to 4MHz (default) or some other value.

Information:

The setfreq command can be used to change the speed of operation of the microcontroller from 4MHz to 8MHz (or other value). 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).

On devices with fixed external resonators this command cannot be used - the value of the external resonator must be changed to alter the clock frequency.

Note that the X2parts have an internal x4 PLL inside the chip. This multiplies the external resonator speed by 4. Therefore the external resonator value to be used is 1/4 of the desired final speed (ie in mode em40 use an external 10MHz resonator, for em16 use a 4MHz resonator).

The change occurs immediately. All programs default to m4 (4MHz) if a setfreq command is not used.

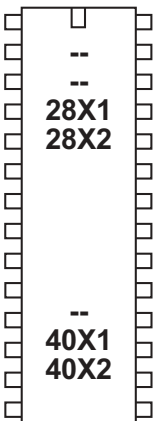
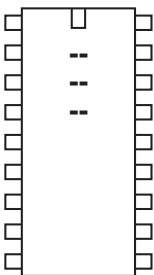
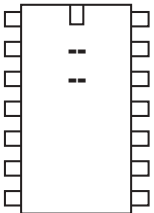
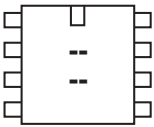
Note that the Programming Editor only supports certain frequencies for new program downloads. If your chip is running at a different frequency the X1 and X2 parts will automatically switch back to internal 4MHz speed to complete the download.

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 (on X1 and X2 parts this is automatic).

Example:

```
setfreq m4           ` setfreq to 4MHz
readtemp 1,b1       ` do command at 4MHz
setfreq m8           ` set freq back to 8MHz
```



settimer

Syntax:

SETTIMER OFF

SETTIMER preload

SETTIMER COUNT preload

- preload is the constant/variable that selects the appropriate timing. For convenience timer 1s value constants are predefined in the compiler.

t1s_4	(preload value 49910 - 1 second at 4MHz)
t1s_8	(preload value 34286 - 1 second at 8MHz)
t1s_16	(preload value 3036 - 1 second at 16MHz)

Function:

Configure and start the internal timer / counter.

Information:

The settimer command is used to configure the hardware timer / counter function. The timer function can be used in two way - as an internal timer or as an external counter (input 0 (C.0) only).

Note that the 'debug' command temporarily disables the timer (during the actual variables transmission). Therefore use of the debug command at the same time as the timer will cause false readings.

External Counter

In external counter mode an internal counter register (not accessible to the end user) is incremented on every positive going edge detected on input 0. This pulse counting occurs in the background, so the PICAXE program can perform other tasks at the same time as it is counting (unlike the count command, which stops other processing during the count command time period). When the internal counter register overflows from 65535 to 0, the special 'timer' variable is automatically incremented.

Therefore to increment the timer variable on every 10 external pulses set the preload value to $65536 - 10 = 65526$. After ten pulses the counter register will overflow and hence increment the 'timer' variable. To increment the 'timer' variable on every external pulse simply set the preload value to 65535.

If the timer word variable overflows (ie from 65535 to 0) the timer overflow flag (toflag) is set. The toflag is automatically cleared upon the settimer command, but can also be cleared manually in software via 'let toflag = 0'. If desired an interrupt can be set to detect this overflow by use of the setintflags command.

Example:

```

settimer count 65535    ` settimer to count mode
main:
pause 10000            ` wait 10 seconds, counting pulses
debug                  ` display timer value
goto main              ` loop

```

Internal Timer

In internal timer mode the time elapsed is stored in the word variable 'timer' which can be accessed as if was a normal variable e.g.

```
if timer > 200 then skip
```

When the timer word variable overflows (ie from 65535 to 0) the timer overflow flag (toflag) is set. The toflag is automatically cleared upon the settimer command, but can also be cleared manually via 'let toflag = 0'. If desired an interrupt can be set to detect this overflow by use of the setintflags command.

The period of the timer can be used defined. The timer operates with 'minor ticks' and 'major ticks'. A minor tick occurs every $1/(\text{clock freq} / 256)$ seconds. With a 4MHz resonator this means a minor tick occurs every 64us (32us at 8MHz, 16us at 16MHz, 12.8us at 20MHz). When the minor tick word variable (not accessible by the end user) overflows (from 65535 to 0) a major tick occurs. The major tick increments the timer variable, and so the number of major ticks passed can be determined by reading the 'timer' variable.

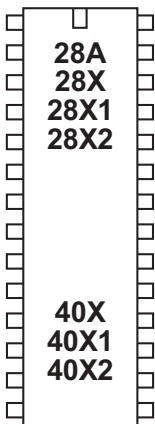
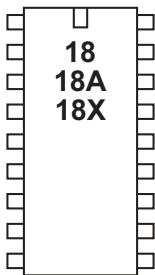
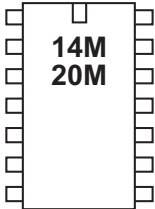
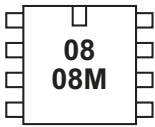
The preload value is used to preload the minor tick variable after it overflows. This means it is not always necessary to wait the full 65536 minor ticks, for instance, if the preload value is set to 60000 you then only have to wait 5536 minor ticks before the major tick occurs.

As an example, assume you wish the timer to increment every second at 4MHz. We know that at 4MHz each minor tick takes 64us and 1 second is equivalent to 1000000 us. Therefore we require 15625 ($1000000 / 64$) minor ticks to give us a 1 second delay. Finally $65536 - 15625 = 49910$, so our preload value become 49910.

Timer cannot be used at the same time as the servo command, as the servo command requires sole use of the timer to calculate the servo pulse intervals.

Example:

```
settimer t1s_4    ` settimer to 1 second ticks at 4MHz
main:
pause 10000       ` wait 10 seconds
debug             ` display timer value
goto main        ` loop
```



shiftin (spiin)

Syntax:

SPIIN *sclk,sdata,mode,(variable {/ bits} {, variable {/ bits}, ...})*

- *sclk* is a variable/constant (0-7) which specifies the i/o pin to use as clock.
- *sdata* is a variable/constant (0-7) which specifies the i/o pin to use as data.
- Mode is a variable/constant (0-7) which specifies the mode:

0	MSBPre_L	(MSB first, sample before clock, idles low)
1	LSBPre_L	(LSB first, sample before clock, idles low)
2	MSBPost_L	(MSB first, sample after clock, idles low)
3	LSBPost_L	(LSB first, sample after clock, idles low)
4	MSBPre_H	(MSB first, sample before clock, idles high)
5	LSBPre_H	(LSB first, sample before clock, idles high)
6	MSBPost_H	(MSB first, sample after clock, idles high)
7	LSBPost_H	(LSB first, sample after clock, idles high)
- Variable receives the data.
- Bits is the optional number of bits to transmit. If omitted the default is 8.

Information:

The spiin (shiftin also accepted by the compiler) command is a 'bit-bang' method of SPI communication on the X1 and X2 parts. For a hardware solution see the 'hshin' command.

By default 8 bits are shifted into the variable. A different number of bits (1 to 8) can be defined via the optional \ bits. Therefore, for instance, if you require to shift in 12 bits, do this as two bytes, one byte shifting 8 bits and the second byte shifting 4 bits. Note that if you are using the LSB first method, the bits are shifted right (in from the left) and so shifting just 4 bits would leave them located in bits 7-4 (not 3-0). With the MSB method the bits are shifted left (in from the right).

When connected SPI devices (e.g. EEPROM) remember that the data-in of the EEPROM connects to the data-out of the PICAXE, and vice versa.

Other PICAXE microcontrollers do not have a direct spiin (shiftin) command. However the same functionality found in other products can be achieved by using the sub procedures listed overleaf.

Affect of increased clock speed:

Increasing the clock speed increases the SPI clock frequency.

Example:

```
spiin 2,1,LSB_Pre_H, (b1 / 8) ` clock 8 bits into b1
```

shiftn/shiftout on PICAXE chips without native commands:

Some PICAXE microcontrollers do not have a shiftn command. However the same functionality found in other products can be achieved by using the sub procedures provided below. These sub-procedures are also saved in the file called shiftn_out.bas in the \samples folder of the Programming Editor software.

To use, simply copy the symbol definitions to the top of your program and copy the appropriate shiftn sub procedures to the bottom of your program.

Do not copy all options as this will waste memory space.

It is presumed that the data and clock outputs (sdata and sclk) are in the low condition before the gosub is used.

BASIC line

```
"shiftn sclk, sdata,mode, (var_in(\bits)) "
```

becomes

```
gosub shiftn_LSB_Pre      (for mode LSBPre)
gosub shiftn_MSB_Pre     (for mode MSBPre)
gosub shiftn_LSB_Post   (for mode LSBPost)
gosub shiftn_MSB_Post   (for mode MSBPost) '
```

```
` ~~~~~ SYMBOL DEFINITIONS ~~~~~
```

```
` Required for all routines. Change pin numbers/bits as required.
` Uses variables b7-b13 (i.e. b7,w4,w5,w6). If only using 8 bits
` all the word variables can be safely changed to byte variables.
`
```

```
`***** Sample symbol definitions *****
```

```
symbol sclk = 5           ` clock (output pin)
symbol sdata = 7         ` data (output pin for shiftout)
symbol serdata = input7 ` data (input pin for shiftn, note input7
symbol counter = b7      ` variable used during loop
symbol mask = w4         ` bit masking variable
symbol var_in = w5       ` data variable used durig shiftn
symbol var_out = w6      ` data variable used during shiftout
symbol bits = 8          ` number of bits
symbol MSBvalue = 128   ` MSBvalue
                        ` (=128 for 8 bits, 512 for 10 bits, 2048 for 12 bits)
```

```

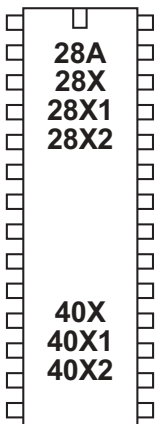
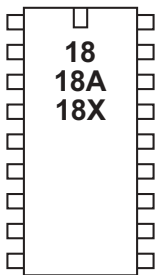
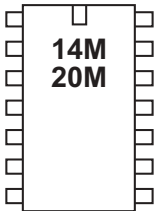
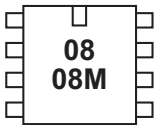
=====
\ ~~~~ SHIFTIM Routines ~~~~
\ Only one of these 4 is required - see your IC requirements
\ It is recommended you delete the others to save space
=====
\ ***** Shiftin LSB first, Data Pre-Clock *****
shiftin_LSB_Pre:
    let var_in = 0
    for counter = 1 to bits \ number of bits
    var_in = var_in / 2 \ shift right as LSB first
    if serdata = 0 then skipLSBPre
    var_in = var_in + MSBValue \ set MSB if serdata = 1
skipLSBPre:
    pulsout sclk,1 \ pulse clock to get next data bit
    next counter
    return

=====
\ ***** Shiftin MSB first, Data Pre-Clock *****
shiftin_MSB_Pre:
    let var_in = 0
    for counter = 1 to bits \ number of bits
    var_in = var_in * 2 \ shift left as MSB first
    if serdata = 0 then skipMSBPre
    var_in = var_in + 1 \ set LSB if serdata = 1
skipMSBPre:
    pulsout sclk,1 \ pulse clock to get next data bit
    next counter
    return

=====
\ ***** Shiftin LSB first, Data Post-Clock ***** \
shiftin_LSB_Post: let var_in = 0
    for counter = 1 to bits \ number of bits
    var_in = var_in / 2 \ shift right as LSB first
    pulsout sclk,1 \ pulse clock to get next data bit
    if serdata = 0 then skipLSBPost
    var_in = var_in + MSBValue \ set MSB if serdata = 1
skipLSBPost:
    next counter
    return

=====
\ ***** Shiftin MSB first, Data Post-Clock *****
shiftin_MSB_Post: let var_in = 0
    for counter = 1 to bits \ number of bits
    var_in = var_in * 2 \ shift left as MSB first
    pulsout sclk,1 \ pulse clock to get next data bit
    if serdata = 0 then skipMSBPost
    var_in = var_in + 1 \ set LSB if serdata = 1
skipMSBPost:
    next counter
    return
=====

```



shiftout (spiout)

Syntax:

SPIOUT *clck,sdata,mode,{data{/ bits}, {data{/ bits},...}*

- *clck* is a variable/constant (0-7) which specifies the i/o pin to use as clock.
- *sdata* is a variable/constant (0-7) which specifies the i/o pin to use as data.
- Mode is a variable/constant (0-3) which specifies the mode:

0	LSBFirst_L	(LSB first, idles low)
1	MSBFirst_L	(MSB first, idles low)
4	LSBFirst_H	(LSB first, idles high)
5	MSBFirst_H	(MSB first, idles high)
- Data is a variable/constant (0-7) that contains the data to send.
- Bits (optional) is the number of bits to transmit. If omitted the default number of bits is automatically set to 8.

Information:

The spiout (shiftout is also accepted by the compiler) command is a bit-bang method of SPI communication on the X1 and X2 parts. For a hardware solution see the 'hspiout' command.

By default 8 bits are shifted out. A different number of bits (1 to 8) can be defined via the optional \ bits. Therefore, for instance, if you require to shift out 12 bits, do this as two bytes, one byte shifting 8 bits and the second byte shifting 4 bits. Note that if you are using the MSB first method, the bits are shifted left (out from the left) and so when shifting just 4 bits they must be located in bits 7-4 (not 3-0). With the LSB method the bits are shifted out from the right.

When connected SPI devices (e.g. EEPROM) remember that the data-in of the EEPROM connects to the data-out of the PICAXE, and vice versa.

Some PICAXE microcontrollers do not have a shiftout command. However the same functionality found in other products can be achieved by using the sub procedures listed below.

Affect of increased clock speed:

Increasing the clock speed increases the SPI clock frequency.

Example:

```
spiout 1,2,LSB_First, (b1 / 8) \ clock 8 bits from b1
```

shiftin/shiftout on PICAXE chips without native commands:

Some PICAXE microcontrollers do not have a shiftin command. However the same functionality found in other products can be achieved by using the sub procedures provided below. These sub-procedures are also saved in the file called shiftin_out.bas in the \samples folder of the Programming Editor software.

To use, simply copy the symbol definitions (listed within the shiftin command) to the top of your program and copy the appropriate shiftout sub procedures below to the bottom of your program.

Do not copy both options as this will waste memory space.

It is presumed that the data and clock outputs (sdata and sclk) are in the low condition before the gosub is used.

BASIC line

“shiftout sclk, sdata,mode, (var_out(\bits))”

becomes

gosub shiftout_LSBFirst (for mode LSBFirst)

gosub shiftout_MSBFirst (for mode MSBFirst)

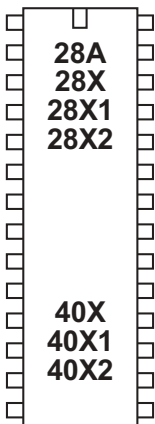
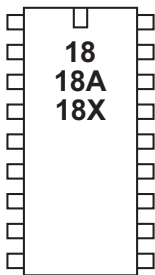
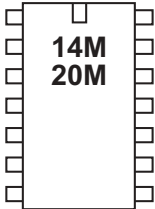
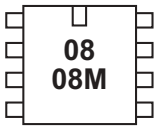
Note the symbol definitions listed in the 'shiftin' command must also be used.

```

=====
\ ***** Shiftout LSB first *****
shiftout_LSBFirst:
    for counter = 1 to bits           \ number of bits
    mask = var_out & 1                \ mask LSB
    low sdata                          \ data low
    if mask = 0 then skipLSB
    high sdata                          \ data high
skipLSB:    pulsout sclk,1            \ pulse clock for 10us
    var_out = var_out / 2              \ shift variable right for LSB
    next counter
    return

=====
\ ***** Shiftout MSB first *****
shiftout_MSBFirst:
    for counter = 1 to bits           \ number of bits
    mask = var_out & MSEValue         \ mask MSB
    low sdata                          \ data low
    if mask = 0 then skipMSB
    high sdata                          \ data high
skipMSB:    pulsout sclk,1            \ pulse clock for 10us
    var_out = var_out * 2              \ shift variable left for MSB
    next counter
    return
=====

```

sleep



Syntax:

SLEEP period

- Period is a variable/constant which specifies the duration of sleep in multiples of 2.3 seconds (0-65535).

Function:

Sleep for some period (multiples of 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.

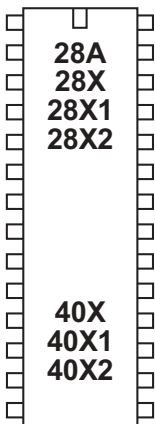
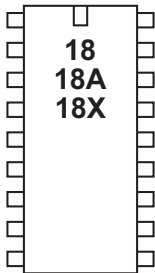
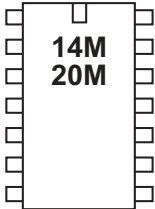
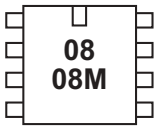
Some PICAXE chips support the disablebod (enablebod) command to disable the brown-out detect function. Use of this command prior to a sleep will considerably reduce the current drawn during the sleep command.

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 (0-7) 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.

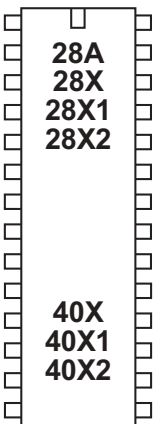
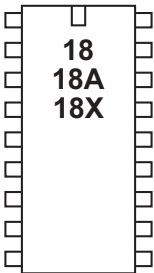
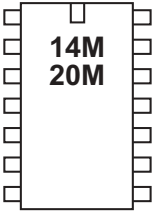
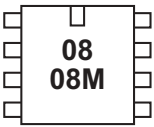
See the tune command for suitable piezo / speaker circuits.

Affect of Increased Clock Speed:

The length of the note is halved at 8MHz and quartered at 16MHz.

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.

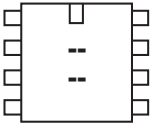
The stop command does not switch off internal timers, and so commands such as servo and pwmout that require these timers will continue to function.

Example:

main:

```
pwmout 1,120,400
```

```
stop
```



swap

Syntax:

SWAP variable1, variable2

Function:

Swap the values between two variables.

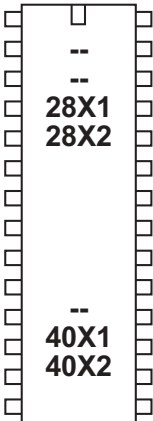
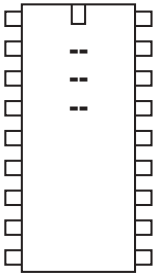
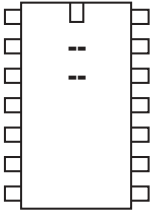
Information:

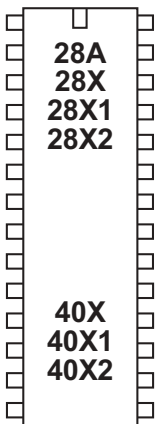
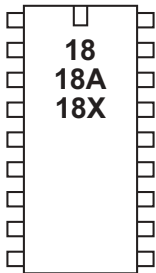
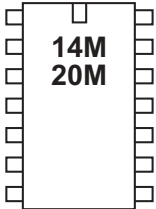
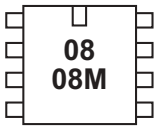
The swap command simply exchanges values between two variables.

Example:

```

    b1 = 5
    b2 = 10
main:
    swap b1,b2
    debug
    pause 1000
    goto main
  
```





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 (0-7) which specifies the i/o pin to use.

Function:

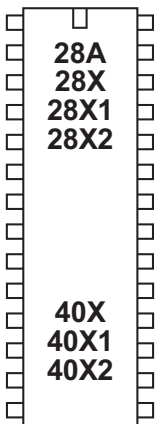
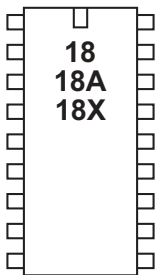
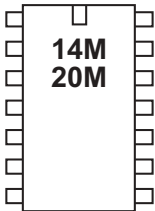
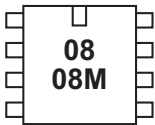
Make pin output high / low.

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.

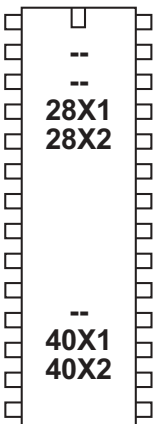
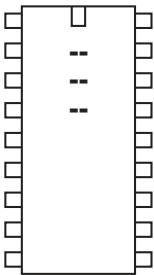
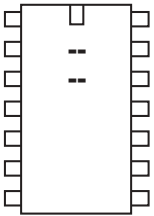
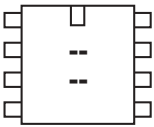
Example:

```

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

```



table

Syntax:

TABLE {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 a lookup table for embedding in the downloaded program.

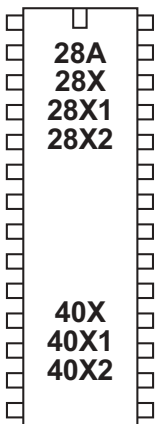
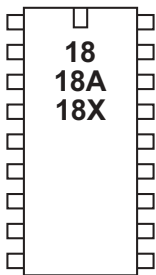
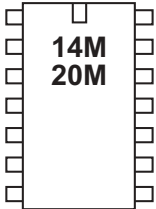
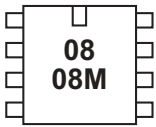
Information:

This is not an instruction, but a method of pre-loading the microcontrollers table. Use of this command reserves 256 bytes within the downloaded program as a lookup table.

Example:

```
TABLE 0,("Hello World")      \ save values in table

main:
  for b0 = 0 to 10            \ start a loop
    readtable b0,b1          \ read value from table
    serout 7,N2400,(b1)      \ transmit to serial LCD module
  next b0                    \ next character
```



toggle



Syntax:

TOGGLE pin,pin,pin...

- Pin is a variable/constant (0-7) which specifies the i/o pin to use.

Function:

Make pin output and toggle state.

Information:

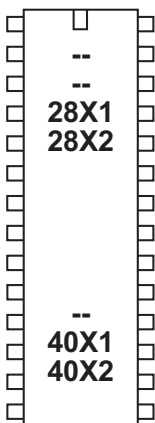
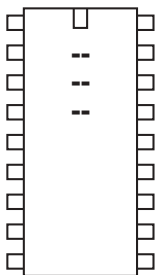
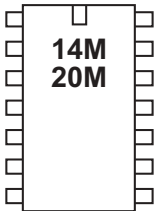
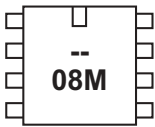
The high command inverts an output (high if currently low and vice versa)

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...) (PICAXE-08M only)

TUNE pin, speed, (note, note, note...)

- pin is a variable/constant (0-7) which specifies the i/o pin to use (not available on PICAXE-08M, which is fixed to output 2).
- LED is a variable/constant (0-3) which specifies if other PICAXE-08M 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 microcontroller with limited memory will never have the quality of commercial playback devices, but the tune command performs remarkably well. Music can be played on economical piezo sounders (as found in musical birthday cards) or on better quality speakers.

The following information gives technical details of the note encoding process. However most users 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. Therefore the technical details are only provided for information only – they are not required to use the Tune Wizard.

Note that the tune command compresses the data, but 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.

All tunes play on a piezo sounder or speaker, connected to the output pin (must be output 2 (leg 5) of the PICAXE-08M). Some sample circuits are shown later in this section.

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.

The sound duration of a quarter beat within the PICAXE is as follows:

$$\text{sound duration} = \text{speed} \times 65.64 \text{ ms}$$

Each quarter beat is also followed by a silence duration as follows,

$$\text{silence duration} = \text{speed} \times 8.20 \text{ ms}$$

Therefore the total duration of a quarter beat is:

$$\begin{aligned} \text{total duration} &= (\text{speed} \times 65.64) \\ &\quad + (\text{speed} \times 8.20) \\ &= \text{speed} \times 73.84 \text{ ms} \end{aligned}$$

Therefore the approximate number of beats per minute (bpm) are:

$$\text{bpm} = 60\,000 / (\text{speed} \times 73.84)$$

A table of different speed values are shown here. This gives a good range for most popular tunes.

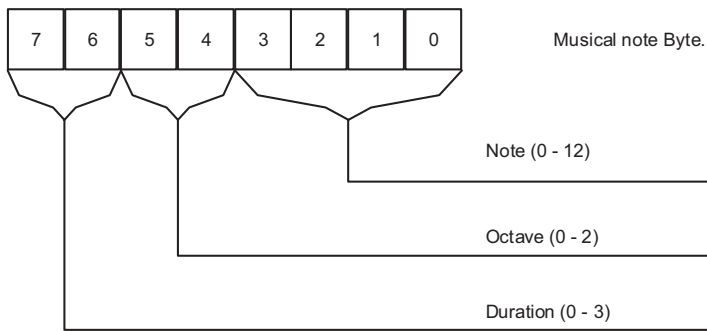
Speed	BPM
1	812
2	406
3	270
4	203
5	162
6	135
7	116
8	101
9	90
10	81
11	73
12	67
13	62
14	58
15	54

Note that within electronic music a note normally plays for 7/8 of the total note time, with silence for 1/8. With the PICAXE the ratio is slightly different (8/9) due to memory and mathematical limitations of the microcontroller.

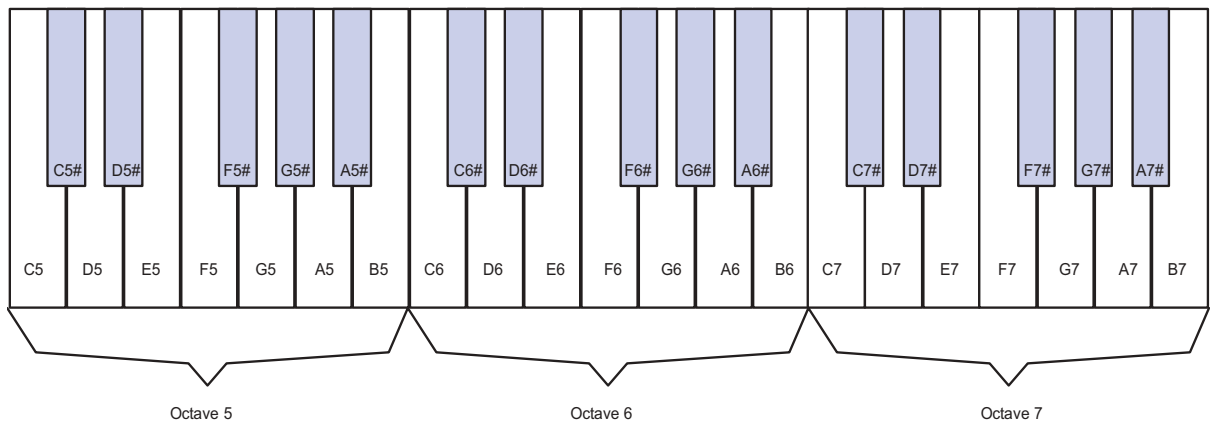
Note Bytes:

Each note byte is encoded into 8 bits as shown. The encoding is optimised to ensure the most common values (1/4 beat and octave 6) both have a value of 00. Note that as the PICAXE also performs further optimisation on the whole tune, the length of the tune will not be exactly the same length as the number of note bytes. 1/16, 1/32 and 'dotted' notes are not supported.

76 = Duration	54 = Octave	3210 = Note
00 = 1/4	00 = Middle Octave (6)	0000 = C
01 = 1/8	01 = High Octave (7)	0001 = C#
10 = 1	10 = Low Octave (5)	0010 = D
11 = 1/2	11 = not used	0011 = D#
		0100 = E
		0101 = F
		0110 = F#
		0111 = G
		1000 = G#
		1001 = A
		1010 = A#
		1011 = B
		11xx = Pause



Piano Representation of Note Frequency



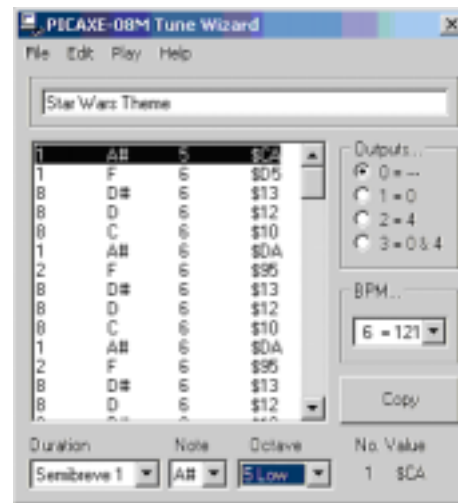
C5 = 262 Hz
 C5# = 277 Hz
 D5 = 294 Hz
 D5# = 311 Hz
 E5 = 330 Hz
 F5 = 349 Hz
 F5# = 370 Hz
 G5 = 392 Hz
 G5# = 415 Hz
 A5 = 440 Hz
 A5# = 466 Hz
 B5 = 494 Hz

C6 = 523 Hz ("Middle C")
 C6# = 554 Hz
 D6 = 587 Hz
 D6# = 622 Hz
 E6 = 659 Hz
 F6 = 698 Hz
 F6# = 740 Hz
 G6 = 784 Hz
 G6# = 831 Hz
 A6 = 880 Hz
 A6# = 932 Hz
 B6 = 988 Hz

C7 = 1047 Hz
 C7# = 1109 Hz
 D7 = 1175 Hz
 D7# = 1245 Hz
 E7 = 1318 Hz
 F7 = 1396 Hz
 F7# = 1480 Hz
 G7 = 1568 Hz
 G7# = 1661 Hz
 A7 = 1760 Hz
 A7# = 1865 Hz
 B7 = 1975 Hz

PICAXE Tune Wizard

The Tune Wizard allows musical tunes to be created for the PICAXE. Tunes can be entered manually using the drop-down boxes if desired, but most users will prefer to automatically import a mobile phone monophonic ringtone. These ringtones are widely available on the internet in RTTTL format (used on most Nokia phones). Note the PICAXE can only play one note at a time (monophonic), and so cannot use multiple note (polyphonic) ringtones.



There are approximately 1000 tunes for free download on the software page of the www.picaxe.co.uk website. Some other possible sources for free ringtones are:

<http://www.ringtonerfest.com/>
<http://www.free-ringtones.eu.com/>
<http://www.tones4free.com/>

To start the Tune Wizard click the PICAXE>Wizard>Tune Wizard menu.

The easiest way to import a ringtone from the internet is to find the tune on a web page. Highlight the RTTTL version of the ringtone in the web browser and then click Edit>Copy. Move back to the Tune Wizard and then click Edit>Paste Ringtone.

To import a ringtone from a saved text file, click File>Import Ringtone.

Once the tune has been generated, select whether you want outputs 0 and 4 to flash as the tune plays (from the options within the 'Outputs' section).

The tune can then be tested on the computer by clicking the 'Play' menu (if your computer is fitted with soundcard and speakers). The tune played will give a rough idea of how the tune will sound on the PICAXE, but will differ slightly due to the different ways that the computer and PICAXE generate and playback sounds. On older computers the tune generation may take a couple of seconds as generating the tune is very memory intensive.

Once your tune is complete click the 'Copy' button to copy the tune command to the Windows clipboard. The tune can then be pasted into your main program.

Tune Wizard menu items:

File	New	Start a new tune
	Open	Open a previously saved tune
	Save As	Save the current tune
	Import Ringtone	Open a ringtone from a text file
	Export Ringtone	Save tune as a ringtone text file
	Export Wave	Save tune as a Windows .wav sound file
	Close	Close the Wizard
Edit	Insert Line	Insert a line in the tune
	Delete Line	Delete the current line
	Copy BASIC	Copy the tune command to Windows clipboard
	Copy Ringtone	Copy tune as a ringtone to Windows clipboard
	Paste BASIC	Paste tune command into Wizard
	Paste Ringtone	Paste ringtone into Wizard
Play		Play the current tune on the computer's speaker
Help	Help	Start this help file.

Ring Tone Tips & Tricks:

1. After generating 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 simplest, most economical, way to play the tunes is to use a piezo sounder. These are simply connected between the output pin (e.g. pin 2 (leg 5) of the PICAXE-08M) and 0V (see circuits below).

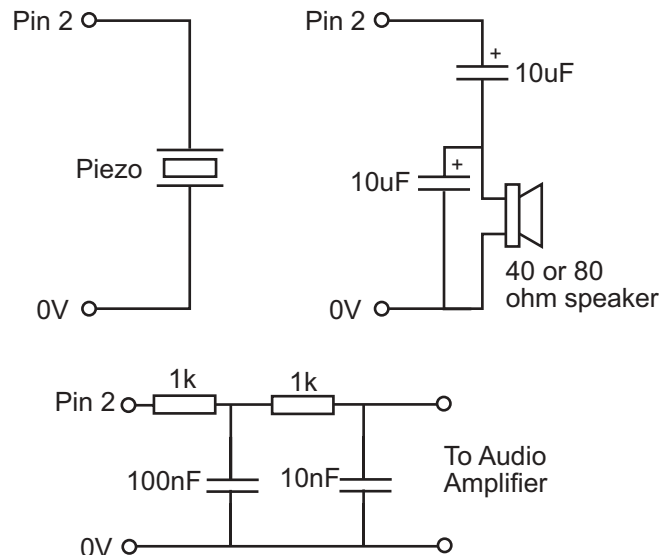
The best piezo sound comes from the 'plastic cased' variants. Uncased piezos are also often used in schools due to their low cost, but the 'copper' side will need fixing to a suitable sound-board (piece of card, polystyrene cup or even the PCB itself) with double sided tape to amplify the sound.

For richer sounds a speaker should be used. Once again the quality of the sound-box the speaker is placed in is the most significant factor for quality of sound. Speakers can be driven directly (using a series capacitor) or via a simply push-pull transistor amplifier.

A 40 or 80 ohm speaker can be connected with two capacitors as shown. For an 8 ohm speaker use a combination of the speaker and a 33R resistor in series (to generate a total resistance of 41R).

The output can also be connected (via a simple RC filter) to an audio amplifier such as the TBA820M.

The sample .wav sound files in the \music sub-folder of the Programming Editor software are real-life recordings of tunes played (via a speaker) from the microcontroller chip.



Ringing 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

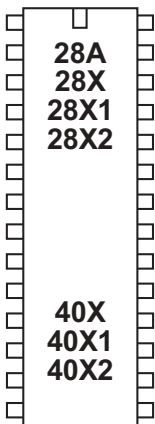
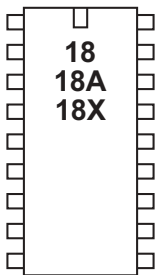
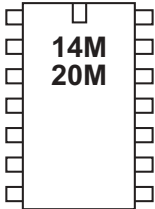
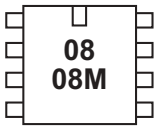
<note> :=
"C" |
"C#" |
"D" |
"D#" |
"E" |
"F" |
"F#" |
"G" |
"G#" |
"A" |
"A#" |
"B" |           ; "H" can also be used      PICAXE exports using B
"P" |           ; pause

<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:

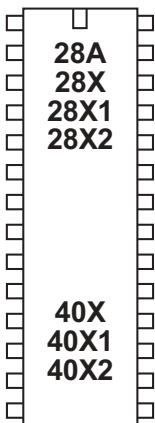
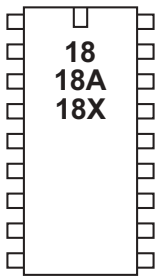
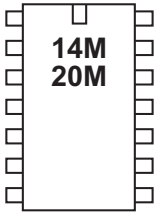
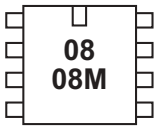
Pause for some time in whole seconds.

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 and 18 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-08, 18	0 to (127 - number of used bytes)
PICAXE-08M, 14M, 20M	0 to (255 - number of used bytes)

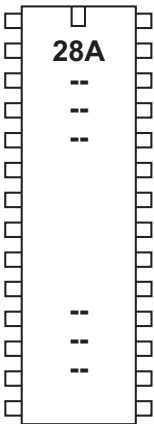
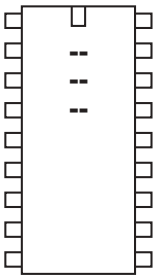
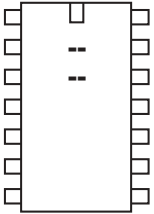
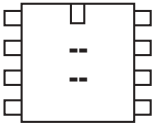
With the following microcontrollers the data memory is completely separate from the program and so no conflicts arise. The number of bytes available varies depending on microcontroller type as follows.

PICAXE-28, 28A	0 to 63
PICAXE-28X, 40X	0 to 127
PICAXE-18A, 18X, 28X1, 28X2, 40X1, 40X2	0 to 255

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 b1
  next b0              \ next loop
```



writemem

Syntax:

WRITEMEM location,data

- 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.

Function:

Write FLASH program memory byte data into location.

Information:

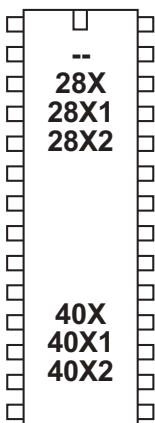
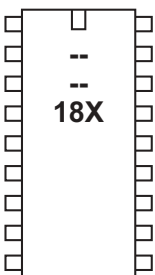
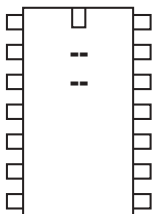
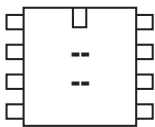
The data memory on the PICAXE-28A is limited to only 64 bytes. Therefore the writemem command provides an additional 256 bytes storage in a second data memory area. This second data area is not reset during a download.

This command is not available on the PICAXE-28X as a larger i2c external EEPROM can be used.

The writemem command is byte wide, so to write a word variable two separate byte write commands will be required, one for each of the two bytes that makes the word (e.g. for w0, read both b0 and b1).

Example:

```
main:
  for b0 = 0 to 255      \ start a loop
    serin 6,T2400,b1    \ receive serial value
    writemem b0,b1      \ write value into b1
  next b0                \ next loop
```



writei2c

Syntax:

WRITEI2C location,(variable,...)

WRITEI2C (variable,...)

- Location is a variable/constant specifying a byte or word address.
- Variable(s) contains the data byte(s) to be written.

Function:

Write i2c location.

For X1 and X2 parts the *hi2cout* command is recommended instead.

Information:

Use of i2c parts is covered in more detail in the separate 'i2c Tutorial' datasheet.

This command is used to write byte data to an i2c device. Location defines the start address of the data to be written, although it is also possible to write more than one byte sequentially (if the i2c device supports sequential writes).

Location must be a byte or word as defined within the *i2cslave* command. An *i2cslave* command must have been issued before this command is used.

Example:

```
; Example of how to use DS1307 Time Clock
; Note the data is sent/received in BCD format.
; Note that seconds, mins etc are variables that need
; defining e.g. symbol seconds = b0 etc.

' set DS1307 slave address
  i2cslave %11010000, i2cslow, i2cbyte

'write time and date e.g. to 11:59:00 on Thurs 25/12/03
start_clock:
  let seconds = $00 ' 00 Note all BCD format
  let mins    = $59 ' 59 Note all BCD format
  let hour    = $11 ' 11 Note all BCD format
  let day     = $03 ' 03 Note all BCD format
  let date    = $25 ' 25 Note all BCD format
  let month   = $12 ' 12 Note all BCD format
  let year    = $03 ' 03 Note all BCD format
  let control = %00010000 ' Enable output at 1Hz

  writei2c 0,(seconds,mins,hour,day,date,month,year,control)
end
```

Additional Reserved Keywords

In addition to the command names (see index on page 1-3), the following are also reserved keywords within the compiler. These words may not be used as labels or symbols within a program.

a, and, andnot
 b, b0 -b63, bit0 - bit15, byte
 b300_4, b600_4, b1200_4, b2400_4, b4800_4, b9600_4, b19200_4
 b300_8, b600_8, b1200_8, b2400_8, b4800_8, b9600_8, b19200_8
 b300_16, b600_16, b1200_16, b2400_16, b4800_16, b9600_16, b19200_16
 b300_20, b600_20, b1200_20, b2400_20, b4800_20, b9600_20, b19200_20
 c, cls, cr
 d, dirs, dir0 - dir7
 flags, flag0-flag7
 hi2cflag, hi2clast, hserinflag, hserptr
 i2cfast, i2cfast8, i2cfast16, i2cslow, i2cslow8, i2cslow16, i2cmaster
 inputa, infra, input0-input7, is
 lf, lsb_first
 keyvalue, k31, k125, k500
 m1, m2, m4, m8, m16, m32, m40, msb_first
 n300, n600, n1200, n2400, n4800
 n300_4, n600_4, n1200_4, n2400_4, n4800_4
 n300_8, n600_8, n1200_8, n2400_8, n4800_8, n9600_8
 n300_16, n600_16, n1200_16, n2400_16, n4800_16, n9600_16
 on, off, or, ornot, outpin0 - outpin7, output0-output7
 pin0 - pin7, port, pot
 pwmfull, pwmsingle, pwmhald, pwmllll, pwmlhhl, pwmlhlh, pwmhhhh
 ptr0-ptr7, ptr, @ptr, @ptrinc, @ptrdec
 spifast, spimedium, spislow
 to, toflag, then, timer, t300, t600, t1200, t2400, t4800
 t300_4, t600_4, t1200_4, t2400_4, t4800_4
 t300_8, t600_8, t1200_8, t2400_8, t4800_8, t9600_8
 t300_16, t600_16, t1200_16, t2400_16, t4800_16, t9600_16
 until
 w0-w13, while, word
 x, x2, xnor, xor, xornot

Manufacturer Website:

Main website: www.picaxe.co.uk
Forum: www.picaxeforum.co.uk
VSM Simulator: www.picaxeism.com

PICAXE products are developed and distributed by
Revolution Education Ltd
<http://www.rev-ed.co.uk/>

Trademark:

PICAXE® is a registered trademark licensed by Microchip Technology Inc.
Revolution Education is not an agent or representative of Microchip
and has no authority to bind Microchip in any way.

Acknowledgements:

Revolution Education would like to thank the following:
Clive Seager
John Bown
LTScotland
Higher Still Development Unit
UKOOA