

Índice

Instruções PBasic – Comandos e Sintaxe	1
Introdução	1
Modo BASIC	1
Modo Estendido	1
Modo PICAXE	1
Modo Assembler	1
Operadores Matemáticos	2
LABELS.....	2
Espaço em branco	2
CÓMENTÁRIOS.....	2
CONSTANTES	3
SÍMBOLOS	3
VARIÁVEIS.....	4
Variáveis de usos gerais.....	4
Variáveis de armazenamento	4
Variáveis para funções especiais (SFRs)	5
INSTRUÇÕES	6
branch	6
button	7
count	8
debug.....	9
dec.....	9
do...loop	10
eeprom	11
end	12
exit	13
for ... next.....	13
gosub.....	14
goto	15
high	16
high portc	16
if ... then \ elseif...then \ else \ endif	17
if ... then {goto}	18
if ... and/or... then {goto}	18
if ... then exit	19
if ... and/or... then exit	19
if ... then gosub.....	20
if... and/or... then gosub	20
inc	21
let	22
Funções matemáticas suportadas	23
let dirs =	24
let dirsc =	24
let pins =	25
let pinsc =.....	25
lookdown	26
lookup	26
low	27
nap	28
on...goto	29
on...gosub.....	29
pause	30
pulsin.....	31
pulsout.....	32

pwmout.....	32
random.....	34
readadc.....	35
readadc10.....	35
read.....	36
return.....	37
select case \ case \ else \ endselect.....	38
serin.....	39
serout.....	40
sertext.....	41
servo.....	42
setint.....	43
sleep.....	45
sound.....	45
stop.....	46
symbol.....	47
toggle.....	48
write.....	48

Instruções PBasic – Comandos e Sintaxe

Introdução

A secção seguinte descreve a sintaxe da linguagem BASIC suportada pelo Editor de Programas da PICAXE. Como o software Editor de Programas suporta uma grande variedade de dispositivos, alguns comandos são apenas suportados nalguns modos de funcionamento. Os quatro modos de funcionamento são:

Modo BASIC

Este modo corresponde à sintaxe original PBASIC™ para os dispositivos BASIC Stamp™ 1, incluindo o Stamp Microcontroller e o Stamp Controller.

Modo Extendido

Este modo suporta os mesmos dispositivos que o modo anterior, mas possui um conjunto de “pseudo” instruções adicionais (por ex: `if input0 is on then...` em vez de `if pin0=1 then...`).

Modo PICAXE

Estas são as instruções suportadas pela gama de microcontroladores PICAXE.

Modo Assembler

Estas são as instruções que podem ser convertidas para código assembly através do comando *automatic sequential* BASIC to Assembler. Veja o *datasheet* sobre o assembler para mais informações.

A parte restante desta secção está dividida nos seguintes tópicos:

- Labels (Etiquetas)
- Comentários
- Constantes
- Símbolos
- Variáveis
- Comandos (instruções)

Operadores Matemáticos

Veja na instrução “let” os detalhes sobre os operadores matemáticos suportados.

LABELS

As *labels* (etiquetas) são usadas como marcadores em todo o programa. As *labels* são usadas para marcar uma posição para onde “saltar” no programa através de uma instrução *goto*, *gosub* ou outra instrução. Uma *label* pode ser qualquer palavra (não reservada) e pode conter dígitos e o carácter *underscore* (_). As *labels* devem ter como carácter inicial uma letra (não um dígito), e são definidas com o sinal dois-pontos (:) a seguir ao nome. O sinal não é necessário quando a *label* faz parte integrante de instruções.

O compilador não é case-sensitive (sensível a maiúsculas), pelo que podem ser usadas indiscriminadamente maiúsculas e minúsculas.

Exemplo:

ciclo:

```
high 1      ; liga a saída 1
pause 500   ; espera de 5 segundos
low 1       ; desliga a saída 1
pause 500   ; espera de 5 segundos
goto ciclo  ; salto para o início
```

Espaço em branco

Whitespace (espaço em branco) é o termo utilizado pelos programadores para definirem a área em branco na impressão de um programa. Nela se incluem os espaços, as tabulações e as linhas vazias. Qualquer uma delas pode ser utilizada no programa para o tornar mais compreensível e facilitar a leitura.

Convencionou-se colocar as *labels* encostadas à esquerda. Todas as outras instruções devem ser espaçadas através da tecla de tabulação. Esta convenção torna o programa mais fácil de ler e de seguir.

COMENTÁRIOS

Os comentários começam por um apóstrofe (‘) ou ponto e vírgula (;) e continuam até ao fim da linha. A instrução REM pode também ser utilizada para inserir comentários.

Exemplos:

```
high 0      ‘coloca pin0 alto
high 0      ;coloca pin0 alto
REM        coloca pin0 alto
```

CONSTANTES

As constantes podem ser declaradas de quatro modos diferentes: decimais, hexadecimais, binárias e ASCII.

Os números decimais são escritos directamente sem qualquer prefixo.

Os números hexadecimais (hex) são precedidos pelo sinal dólar (\$).

Os números binários são precedidos pelo sinal de percentagem (%).

Os valores ASCII são colocados entre plicas (“”).

Exemplos:

100	‘ 100 em decimal
\$64	‘ 64 hex
%01100100	‘ 01100100 binário
“A”	‘ “A” ascii (65)
“Hello”	‘ “Hello” – equivalente a “H”, “e”, “l”, “l”, “o”.
B1 = B0 ^ \$AA	‘ ou exclusivo da variável B0 com AA hex

SÍMBOLOS

Os *símbolos* (*symbol*) podem ser associados a valores constantes, nomes *alias* (alternativos) para variáveis e endereços de programa. Os valores constantes e os nomes *alias* de variáveis são atribuídos fazendo seguir ao nome do símbolo o sinal de igual (=), seguido da variável ou constante.

Os *símbolos* podem utilizar qualquer palavra que não seja reservada (instruções).

Os símbolos podem conter letras e números (flash1, led3, etc.), mas o primeiro carácter é obrigatoriamente uma letra. O uso dos símbolos não aumenta a dimensão do programa e torna-o mais legível.

Os endereços de programa são atribuídos fazendo seguir o símbolo pelo sinal dois pontos (:).

Exemplo:

symbol RED_LED = 7	‘ define um pino de saída
symbol CONTA = B0	‘ define o símbolo de uma variável
let CONTA = 200	‘ carrega a variável com o valor 200
CICLO:	‘ define endereço de programa
high RED_LED	‘ um símbolo endereço termina por dois pontos
pause CONTA	‘ liga a saída 7
low RED_LED	‘ espera 0,2 segundos (200 milisegundos)
pause CONTA	‘ desliga a saída 7
goto CICLO	‘ espera 0,2 segundos (200 milisegundos)
	‘ efectua um salto para o início CICLO (ciclo)

VARIÁVEIS

A memória RAM é utilizada para armazenar dados temporários em variáveis durante a execução do programa. A memória perde todos os dados quando é feito *reset* (reinicialização) ou quando se desliga a alimentação. Existem três tipos de variáveis – de usos gerais, de armazenamento e para fins especiais.

Veja a instrução *let* para mais pormenores sobre as variáveis matemáticas.

Variáveis de usos gerais

Existem 14 variáveis de usos gerais tipo *byte*. Estas variáveis *byte* são designadas b0 a b13. As variáveis *byte* podem guardar números inteiros entre 0 e 255. As variáveis *byte* não podem guardar números negativos ou fraccionários, e produzem “overflow” (transbordo) sem qualquer aviso, se o valor guardado exceder a gama 0-255. Por exemplo, $254 + 3 = 1$ ou $2 - 3 = 255$.

É possível guardar números grandes em duas variáveis *byte* combinadas para constituir uma variável tipo *word*, que é capaz de guardar números inteiros de 0 a 65535. Estas variáveis *word* são designadas de w0 a w6 e são construídas de seguinte modo:

w0 = b1:b0
w1 = b3:b2
w2 = b5:b4
w3 = b7:b6
w4 = b9:b8
w5 = b11:b10
w6 = b13:b12

Portanto o *byte* mais significativo de w0 é b1, e o menos significativo é b0.

Por outro lado, os *bytes* b0 e b1 (w0) estão separados em variáveis *bit*. Estas variáveis *bit* podem ser usadas sempre que se pretender guardar um dado de 1 bit.

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

Pode usar-se uma variável *byte*, *word* ou *bit* em qualquer expressão matemática ou instrução que utilize variáveis. Deve-se contudo ter cuidado com a possível repetição da mesma variável usada individualmente e como parte de outra. Por exemplo, não usar b0 e w0 separadamente no mesmo programa.

Todas as variáveis de usos gerais são postas a 0, quando há uma reinicialização do programa.

Variáveis de armazenamento

As variáveis de armazenamento são localizações de memória adicionais destinadas ao armazenamento temporário de dados *byte*. Estas variáveis não podem ser

utilizadas directamente em operações matemáticas, mas podem guardar temporariamente valores tipo byte através das instruções *peek* e *poke*.
Veja as instruções *peek* e *poke* para mais informações.

O Picaxe-28X possui 112 variáveis deste tipo, a saber:
80 a 127 (\$50 a \$7F) e 192 a 239 (\$C0 a \$FF).

Variáveis para funções especiais (SFRs)

pins = o porto de entrada quando se faz leitura do porto
pins = o porto de saída quando se escreve no porto
Repare que *pins* é uma “pseudo” variável que pode ser utilizada quer na leitura quer na escrita do porto.

Quando usada à esquerda de uma declaração, *pins* aplica-se ao porto de “saída”, ex:

```
let pins = %11000011
```

vai ligar as saídas 7,6,1 e 0 deixando as outras desligadas.

Quando se usa *pins* à direita de uma declaração, estamos a referir-nos ao porto de entrada, ex:

```
let b1 = pins
```

vai guardar em b1 a leitura do porto de entrada.

Acrescente-se que a declaração

```
let pins = pins
```

é válida e significa fazer o porto de saída igual ao porto de entrada.

A variável *pins* pode ser subdividida em variáveis bit individuais para leitura dos pinos individualmente, através da instrução `if ... then`.

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

INSTRUÇÕES

branch

Sintaxe:

BRANCH *offset*, (**endereço0**, **endereço1**, ..., **endereçoN**)

- *Offset* é uma variável/constante que especifica qual o endereço a usar (0-N).
- Endereços são *labels* que especificam para onde ir.

Função:

Salta para o endereço especificado pelo *offset* (se dentro do intervalo).

Informação:

Esta instrução permite saltar para diferentes posições no programa dependendo do valor da variável “*offset*”. Se o *offset* tiver o valor 0, o programa saltará para o endereço0, se o *offset* tiver o valor 1, irá ser executado a partir do endereço1, etc. Se o *offset* exceder o número de endereços, a instrução é ignorada e o programa continua na linha seguinte.

Esta instrução é idêntica à instrução *on ... goto*.

Exemplo:

```
reset:
    let b1 = 0
    low 0
    low 1
    low 2
    low 3
princ:
    let b1 = b1 + 1
    if b1 > 3 then reset
    branch b1, (btn0,btn1,btn2,btn3)

btn0:   high 0
        goto princ

btn1:   high 1
        goto princ

btn2:   high 2
        goto princ

btn3:   high 3
        goto princ
```

button

Sintaxe:

BUTTON pino,downstate,delay,rate,bytevariable,targetstate,endereço

- Pino é uma variável/constante (0-7) que especifica o pino de i/o a usar.
- Downstate é uma variável/constante (0 ou 1) que especifica o nível lógico lido quando o botão é pressionado.
- Delay é uma variável/constante (0-255) que especifica o tempo antes de repetição, se a instrução *button* for utilizada dentro de um ciclo repetido.
- Rate é uma variável/constante (0-255) que especifica a taxa de auto-repetição em ciclos repetidos da instrução *button*.
- Bytevariable é a variável de trabalho. Deve ser inicializada a 0 antes de ser utilizada pela instrução *button* pela primeira vez.
- Targetstate é uma variável/constante (0 ou 1) que especifica o nível lógico (0 = não premido, 1 = premido) em que o botão deve estar para que ocorra um salto.
- Endereço é uma *label* que especifica o local do programa para onde este deve saltar se o botão estiver no estado activo.

Função:

Faz o *debounce* (limpa os ressaltos) de um interruptor (botão de pressão), auto-repete, e salta para a posição de memória indicada se o interruptor estiver premido.

Informação:

Quando se activa um interruptor mecânico os seus contactos metálicos não se fecham imediatamente, antes ressaltam um contra o outro, um certo número de vezes antes de estabilizarem. Isto pode levar o microcontrolador a registar vários contactos para uma única acção. Um modo simples de ultrapassar este problema consiste em colocar uma pausa (por ex. *pause 10*) no programa, o que dá tempo ao interruptor para estabilizar.

Em alternativa pode ser usada a instrução *button* para isso. Quando a instrução *button* é executada, o microcontrolador vai ver se a condição 'downstate' se verifica. Se for verdadeira, o interruptor é filtrado da trepidação (*debounce*) e o programa salta para o endereço definido, no caso de 'targetstate' = 1. Se 'targetstate' = 0 o programa continua.

Se a instrução *button* estiver dentro de um ciclo, a próxima vez que a instrução for executada a condição 'downstate' é de novo verificada. Se a condição for ainda verdadeira, a variável 'bytevariable' é incrementada. Isto pode verificar-se o número de vezes necessário para que o valor de 'bytevariable' seja igual ao de 'delay'. Nesta altura dá-se um salto para o endereço definido, se 'targetstate' = 1. A *bytevariable* é então reiniciada a 0 e todo o processo se repete, mas desta vez o salto para o endereço só se dá quando o valor de 'bytevariable' for igual a 'rate'.

Note que a instrução *button* deve ser utilizada dentro de ciclos. A instrução não pára a execução do programa pelo que só verifica a situação do interruptor quando o programa executa a instrução.

Exemplo:

```
princ:
  button 0,0,200,100,b2,0,cont    ‘ salta para cont só quando pin0 = 0
  toggle 1                          ‘ caso contrário muda o valor do pino 1
  goto princ

cont: etc.
```

count

Sintaxe:

COUNT pino, período, var

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.
- Período (1 – 65535) ms
- Variável recebe o resultado (normalmente uma variável *word*) (0-65535).

Função:

Conta impulsos entrados no pino e guarda a contagem na variável.

Informação:

Count verifica o estado do pino de entrada e conta o número de transições de baixo para alto. A 4 MHz (frequência típica do relógio do PICAXE 28X) o pino de entrada é verificado em cada 20 us, pelo que a frequência máxima dos impulsos que podem ser contados é de 25 kHz, assumindo um ciclo de trabalho de 50% (isto é, iguais tempos *on* e *off*).

Tome cuidado com os interruptores mecânicos, que podem produzir múltiplos contactos (impulsos) em cada vez que são premidos, pois existe um fenómeno de ressaltos mecânicos no contacto. Este efeito pode ser ultrapassado através da instrução *button* (ver Manual original) ou da introdução de uma pequena pausa após a leitura do pino.

Exemplo:

```
ciclo:
  count 1, 5000, w1    ‘ conta impulsos cada 5 segundos
  debug w1              ‘ mostra o resultado no monitor do computador
  goto ciclo            ‘ volta ao início
```

debug

Sintaxe:

DEBUG {var}

- Var é um valor de variável opcional (por ex. b1). O seu valor não tem significado e é incluído apenas para compatibilidade com programas mais antigos.

Função:

Mostra informação na janela *debug* do PC sobre a execução do programa.

Informação:

A instrução *debug* faz “upload” dos valores de todas as variáveis através do cabo série de comunicação com o PC, mostrando esses valores numa janela do ecrã. Isso permite ao programador a detecção de erros e o ajuste de parâmetros do programa. Note que a instrução *debug* faz “upload” de uma grande quantidade de dados pelo que atrasa bastante a execução do programa. Deve-se portanto retirar esta instrução do programa após a depuração e ajustes.

Para apresentar no ecrã mensagens de utilizador utilize a instrução *sertxd*.

Exemplo:

ciclo:	debug b1	‘ mostra valor
	let b1 = b1 + 1	‘ incrementa o valor de b1
	pause 500	‘ espera 0,5 segundos
	goto ciclo	‘ volta para o início

dec

Sintaxe:

DEC var

- var é a variável a decrementar

Função:

Decrementa (subtrai 1 unidade a) o valor da variável.

Informação:

Esta instrução é uma abreviação de ‘let var = var - 1’

Exemplo:

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

do...loop

Sintaxe:

```
DO
{codigo}
LOOP UNTIL/WHILE VAR ?? COND
```

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

```
DO UNTIL/WHILE VAR ?? COND
{codigo}
LOOP
```

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

- Var é o valor a testar.
- Cond é a condição.

?? pode ser qualquer uma das seguintes condições

=	igual a
<>	não igual a (diferente)
!=	não igual a (diferente)
>	maior que
>=	maior que ou igual a
<	menor que
<=	menor que ou igual a

Função:

Repete o ciclo enquanto a condição for verdadeira (*while*) ou falsa (*until*)

Informação:

Esta estrutura cria um ciclo que permite que o código seja repetido enquanto, ou até, que uma determinada condição se verifique. A condição pode estar na linha 'do' (a condição é testada antes da execução do código) ou na linha 'loop' (a condição é testada depois da execução do código).

A instrução *exit* pode ser usada para sair em qualquer altura do ciclo *do...loop*.

Exemplo:

```
do
```

```

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

```

EEPROM

Sintaxe:

DATA {localização}, (dado, dado,...)
EEPROM {localização}, (dado, dado,...)

- Localização é uma constante opcional (0-255) que especifica onde se inicia o carregamento dos dados na *EEPROM*. Se não for especificada a localização, o armazenamento continua a partir do último endereço utilizado. Se nenhuma localização tiver sido previamente utilizada, o armazenamento começa do endereço 0.
- Dado são constantes (0-255), que se querem armazenar na *EEPROM*.

Função:

Carrega posições de memória da EEPROM com dados. Se não for usada a instrução EEPROM os valores das variáveis são automaticamente colocados a 0. As palavras-chave DATA e EEPROM têm significado idêntico.

Informação:

Não se trata de uma instrução, antes um meio de carregar previamente localizações de memória da EEPROM. Não afecta a dimensão do programa. No PICAXE 28X esta instrução é mapeada para a memória FLASH separada que possui 128 bytes (0-127).

Exemplo:

```

EEPROM 0, ("Hello World")      ' guarda os valores ASCII na EEPROM

main:

for b0 = 0 to 10                ' inicia o ciclo
  read b0, b1                   ' lê valor da EEPROM e guarda na var b1
  serout 7, T2400, (b1)         ' transmite em série para o display LCD
next b0                          ' próximo carácter

```

end

Sintaxe:

END

Função:

Fica em espera até que a alimentação seja novamente ligada ou o PC ligue. A potência consumida é reduzida ao mínimo absoluto (assumindo que não existem cargas ligadas nas saídas).

Informação:

A instrução *end* coloca o microcontrolador num estado de baixo consumo depois da execução de um programa. Note que como o compilador coloca sempre esta instrução no fim do programa, raramente se usa esta instrução.

A instrução *end* desliga os temporizadores internos, pelo que instruções como *servo* ou *pwmout* que necessitam de usar estes temporizadores não funcionam após a execução desta instrução.

Se não pretender que seja executada a instrução *end*, coloque uma instrução *stop* no fim do programa. A instrução *stop* não coloca o microcontrolador em modo de baixa potência.

A principal utilização da instrução *end* é como separador do programa principal em relação às subrotinas (procedimentos). Isso garante que o programa não entra acidentalmente nas subrotinas.

Exemplo:

```
ciclo:
    let b2 = 15           ' atribui 15 ao valor de b2
    pause 2000           ' espera 2 segundos (2000 milisegundos)
    gosub flsh           ' chama procedimento flsh
    let b2 = 5           ' atribui 5 ao valor de b2
    pause 2000           ' espera 2 segundos (2000 milisegundos)
    gosub flsh           ' chama procedimento flsh
    end                  ' impede entrada acidental no procedimento

flsh:
    for b0 = 1 to b2     ' define b2 como número de ciclos
        high 1           ' liga a saída 1
        pause 500        ' espera 0,5 segundos
        low 1            ' desliga a saída 1
        pause 500        ' espera 0,5 segundos
    next b0              ' fim do ciclo
    return               ' retorno do procedimento
```

exit

Sintaxe:

EXIT

Função:

A instrução Exit é utilizada para terminar imediatamente um ciclo de programa do...loop ou for...next.

Informação:

É equivalente a um 'goto linha seguinte ao loop'.

Exemplo:

```
main:
  do                               ' inicio do loop
    if b1 = 1 then
      exit
    end if
  loop                             ' loop
```

for ... next

Sintaxe:

**FOR variável = inicio TO fim {STEP {-} incremento}
NEXT {variável}**

- Variável vai ser usada como um contador
- Início é o valor inicial da variável
- Fim é o valor final da variável
- Incremento é um valor opcional que se sobrepõe ao valor de incremento normal do contador (+1). Se o incremento for precedido de um '-', será considerado que Fim é menor que Início e, portanto, o valor de incremento é subtraído cada vez que o ciclo se realiza.

Função:

Define um ciclo repetido FOR-NEXT.

Informação:

Os ciclos *for... next* são utilizados para repetir secções de código um certo número de vezes. Quando se usa uma variável byte, o ciclo repete-se até 255 vezes. Cada

vez que a linha *next* é encontrada, o valor da variável é incrementado (ou decrementado) do valor definido por *step* (+1 por omissão). Quando o valor final é ultrapassado o ciclo pára e o fluxo do programa continua a partir da linha seguinte à instrução *next*.

Os ciclos *for...next* podem ser encadeados até 8 níveis de profundidade.

Exemplo:

ciclo:

```
for b0 = 1 to 20 ' define um ciclo de 20 vezes
  high 1        ' liga a saída 1
  pause 500     ' espera 0,5 segundos
  low 1         ' desliga a saída 1
  pause 500     ' espera 0,5 segundos

next b0        ' salta para o início, incrementando b0 de +1,
               ' até que b0 = 20

pause 2000     ' espera 2 segundos
goto ciclo     ' salto para o início
```

gosub

Sintaxe:

GOSUB endereço

- Endereço é uma *label* (etiqueta) que especifica o endereço.

Função:

Salta para a subrotina (procedimento) localizado no endereço, regressando quando encontra a instrução *return*. São permitidas até 16 GOSUBs, podendo ser aninhadas até 4 níveis.

Informação:

A instrução *gosub* (ir para um procedimento), é um salto temporário para uma secção separada do código, de onde regressará, através da instrução *return*. Cada instrução *gosub*, deve ter uma instrução *return* correspondente.

Não deve confundir esta instrução com a instrução *goto*, que é um salto incondicional para uma nova localização no programa.

Os procedimentos ou subrotinas, são largamente utilizados na programação para reduzir o tamanho dos programas, usando secções de código que se repetem num único procedimento. A passagem de valores para o procedimento por variáveis, permite repetir a mesma secção de código a partir de várias localizações do programa.

Exemplo:

```
ciclo:
  let b2 = 15           ' atribui 15 ao valor de b2
  pause 2000           ' espera 2 segundos (2000 milisegundos)
  gosub flsh           ' chama procedimento flsh
  let b2 = 5           ' atribui 5 ao valor de b2
  pause 2000           ' espera 2 segundos (2000 milisegundos)
  gosub flsh           ' chama procedimento flsh
end                    ' impede entrada acidental no procedimento
```

```
flsh:
  for b0 = 1 to b2     ' define b2 como número de ciclos
    high 1             ' liga a saída 1
    pause 500          ' espera 0,5 segundos
    low 1              ' desliga a saída 1
    pause 500          ' espera 0,5 segundos
  next b0              ' fim do ciclo
return                ' retorno do procedimento
```

goto

Sintaxe:

GOTO endereço

- Endereço é uma label (etiqueta) que especifica para onde saltar.

Função:

O programa passa a ser executado a partir do endereço especificado (salto incondicional).

Informação:

A instrução goto é um salto permanente para uma nova secção do programa. O local de salto é assinalado por uma *label*.

Exemplo:

```
ciclo:
  high 1               ' liga a saída 1
  pause 500            ' espera 0,5 segundos
  low 1                ' desliga a saída 1
  pause 500            ' espera 0,5 segundos
  goto ciclo
```

high

Sintaxe:

HIGH pino

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.

Função:

Coloca o pino no nível lógico alto.

Informação:

A instrução *high* coloca um pino de saída alto (ligado).

Exemplo:

```
ciclo:
    high 1          ' liga a saída 1
    pause 500      ' espera 0,5 segundos
    low 1          ' desliga a saída 1
    pause 500      ' espera 0,5 segundos
    goto ciclo
```

high portc

Sintaxe:

HIGH PORTC pino

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.

Função:

Coloca alto o pino de saída do portc indicado.

Informação:

A instrução *high* liga uma saída do portc.

Exemplo:

```
ciclo:
    high portc 1   ' liga a saída 1
    pause 5000    ' espera 5 segundos
    low portc 1   ' desliga a saída 1
    pause 5000    ' espera 5 segundos
    goto ciclo    ' volta ao início
```

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

Sintaxe:

```
IF variável ?? valor (AND/OR variável ?? valor ...) THEN
{código}
ELSEIF variável ?? valor {AND/OR variável ?? valor...} THEN
{código}
ELSE
{código}
ENDIF
```

- Variável (s) é comparada com o valor(s).
- Valor é uma variável/constante.
- Endereço é uma *label* (etiqueta) que especifica o endereço para onde saltar se a condição se verificar (for verdadeira).

?? pode ser qualquer uma das seguintes condições

=	igual a
<>	não igual a (diferente)
!=	não igual a (diferente)
>	maior que
>=	maior que ou igual a
<	menor que
<=	menor que ou igual a

Função:

Compara e executa condicionalmente secções de código.

Informação:

A instrução *if ... then \ elseif \ else \ endif* é usada para testar variáveis de pinos de entrada (ou variáveis de uso geral) perante certas condições. Se essas condições se verificam essa secção de código do programa é executada, e o programa continua na posição seguinte à instrução *endif*. Se a condição não se verifica, a instrução é ignorada e o programa salta directamente para a instrução *elseif* ou *else* seguinte.

A secção de código *else* só é executada se nenhuma das condições *if* ou *elseif* forem verdadeiras.

No caso de utilização com entradas, deve usar-se a variável de entrada (pin1, pin2, etc.) e não o nome do pino (1, 2, etc.), isto é a linha deve ser “if pin1 = 1 then...” e não “if 1 = 1 then...”.

Note que

```
if b0 > 1 then (goto) label      '(estrutura em uma linha única)
```

```
if b0 > 1 then gosub label      '(estrutura em uma linha única)
if b0 > 1 then...else...endif  '(estrutura multi-linhas)
```

são três estruturas completamente diferentes e não podem ser combinadas. Portanto, a linha seguinte é inválida e tenta combinar duas estruturas diferentes.

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

Estas linhas são inválidas pois o compilador não sabe qual a estrutura que está a tentar utilizar:

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

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

Para concretizar esta estrutura a linha deve ser reescrita como

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

O carácter : separa as secções, fornecendo sintaxe correcta ao compilador.

if ... then {goto}

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

Sintaxe:

IF variável ?? valor {AND/OR variável ?? valor...} THEN endereço

- Variável (s) é comparada com o valor(s).
- Valor é uma variável/constante.
- Endereço é uma *label* (etiqueta) que especifica o endereço para onde saltar se a condição se verificar (for verdadeira).

?? pode ser qualquer uma das seguintes condições

=	igual a
<>	não igual a (diferente)
!=	não igual a (diferente)
>	maior que
>=	maior que ou igual a
<	menor que
<=	menor que ou igual a

Função:

Compara e efectua salto condicional para uma nova posição no programa.

Informação:

A instrução *if ... then* é usada para testar variáveis de pinos de entrada (ou variáveis de uso geral) perante certas condições. Se essas condições se verificam o fluxo do programa continua numa nova localização indicada pela *label*. Se a condição não se verifica, a instrução é ignorada e o programa continua na linha seguinte.

No caso de utilização com entradas, deve usar-se a variável de entrada (pin1, pin2, etc.) e não o nome do pino (1, 2, etc.), isto é a linha deve ser “if pin1 = 1 then...” e não “if 1 = 1 then...”.

A instrução *if ... then* apenas verifica uma entrada na altura em que a instrução é executada. É portanto normal colocar a instrução *if...then* dentro de um ciclo do programa que verifica regularmente a entrada. Para mais detalhes sobre como verificar permanentemente uma condição usando *interrupts*, veja a instrução *setint*.

Exemplo:

Teste de uma entrada num ciclo.

início:

```
if pin0 = 1 then acende ‘ salta para acende se o pin0 estiver alto
goto inicio
```

acende:

```
high 1 ‘ põe alta a saída 1
pause 5000 ‘ espera 5 segundos
low 1 ‘ põe baixa a saída 1
pause 5000 ‘ espera 5 segundos
goto inicio ‘ salto incondicional para o início
```

if ... then exit

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

Sintaxe:

IF variável ?? valor {AND/OR variável ?? valor...} THEN EXIT

- Variável (s) é comparada com o valor(s).
- Valor é uma variável/constante.

?? pode ser qualquer uma das seguintes condições

=	igual a
<>	não igual a (diferente)
!=	não igual a (diferente)
>	maior que
>=	maior que ou igual a
<	menor que
<=	menor que ou igual a

Função:

Compara e sai de um ciclo *do...loop* ou *for...next* condicionalmente.

Informação:

A instrução *if ... then exit* é usada para testar variáveis de pinos de entrada (ou variáveis de uso geral) perante certas condições. Se essas condições se verificam o ciclo actual (*do...loop* ou *for...next*) termina imediatamente.

Pode fazer-se comparações múltiplas combinando ANDs e ORs. Para ver exemplos de uso de AND e OR veja a instrução *if...then goto*.

Exemplo:

Testando uma entrada num ciclo.

```
do
    if pin0 = 1 then exit    ' aguarda a passagem do pin0 a 1
loop
```

if ... then gosub

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

Sintaxe:

IF variável ?? valor {AND/OR variável ?? valor...} THEN GOSUB endereço

- Variável (s) é comparada com o valor(s).
- Valor é uma variável/constante.
- Endereço é uma *label* (etiqueta) que especifica o endereço da subrotina para onde saltar se a condição se verificar (for verdadeira).

?? pode ser qualquer uma das seguintes condições

=	igual a
<>	não igual a (diferente)
!=	não igual a (diferente)
>	maior que
>=	maior que ou igual a

< menor que
<= menor que ou igual a

Função:

Compara e efectua condicionalmente a chamada de uma subrotina.

Informação:

A instrução *if ... then gosub* é usada para testar variáveis de pinos de entrada (ou variáveis de uso geral) perante certas condições. Se essas condições se verificam é chamada uma subrotina. Se a condição não se verifica, a instrução é ignorada e o programa continua na linha seguinte. Qualquer subrotina executada regressa à linha seguinte de onde foi chamada.

No caso de utilização com entradas, deve usar-se a variável de entrada (pin1, pin2, etc.) e não o nome do pino (1, 2, etc.), isto é a linha deve ser “if pin1 = 1 then gosub...” e não “if 1 = 1 then gosub...”.

A instrução *if ... then gosub* apenas verifica uma entrada na altura em que a instrução é executada. É portanto normal colocar a instrução *if...then* dentro de um ciclo do programa que verifica regularmente a entrada.

Pode fazer-se comparações múltiplas combinando ANDs e ORs. Para ver exemplos de uso de AND e OR veja a instrução *if...then goto*.

Exemplo:

Teste de uma entrada num ciclo.

início:

```
if pin0 = 1 then gosub acende    ‘ chama subrotina acende se o pin0 estiver alto
goto inicio
```

acende:

```
high 1                            ‘ põe alta a saída 1
pause 500                        ‘ espera 0.5 segundos
low 1                             ‘ põe baixa a saída 1
pause 500                        ‘ espera 0.5 segundos
return                            ‘ regresso à instrução seguinte à de chamada
```

inc

Sintaxe:

INC var

- var é a variável a incrementar

Função:

Incrementa (adiciona 1 a) o valor da variável.

Informação:

Esta instrução é uma abreviação para 'let var = var + 1'

Exemplo:

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

let

Sintaxe:

{LET} variável = {-} valor ?? valor ...

- Variável sobre a qual se realiza a operação
- Valor(s) são variáveis/constantes que operam sobre a variável

Note que as palavras-chave AND e OR são apenas utilizadas dentro das instruções *if ... then* e não são idênticas aos símbolos matemáticos & e | usados nesta instrução.

Função:

Realiza manipulação da variável (de igual tamanho). As operações matemáticas são realizadas estritamente da esquerda para a direita.

Informação:

O microcontrolador pode manipular matematicamente dados de 16 bits (*words*). Os números válidos são os inteiros de 0 a 65535. Todas as operações matemáticas podem ser também aplicadas a dados de 8 bits (byte) que variam de 0 a 255. O microcontrolador não pode tratar números fraccionários ou negativos.

Contudo, é muitas vezes possível reescrever equações com fracções utilizando inteiros, como por exemplo:

let w1 = w2 / 5.7 não é válido, mas

let w1 = w2 * 10 / 57 é matematicamente igual e já é válido.

Funções matemáticas suportadas

As funções matemáticas suportadas são:

+	; adição
-	; subtracção
*	; multiplicação (devolve a parte baixa do resultado - word)
**	; multiplicação (devolve a parte alta do resultado - word)
/	; divisão (devolve o quociente)
//	; divisão (devolve o resto)
MAX	; máximo - torna menor ou igual ao máximo
MIN	; mínimo - torna maior ou igual ao mínimo
& (ou AND)	; operação lógica AND bit a bit
(ou OR)	; operação lógica OR bit a bit
^ (ou XOR)	; operação lógica XOR bit a bit
&/ (ou NAND)	; operação lógica AND NOT (NAND) bit a bit
/ (ou NOR)	; operação lógica OR NOT (NOR) bit a bit
^/ (ou XNOR)	; operação lógica XOR NOT (XNOR) bit a bit

Não existe a função *shift left* (<<) ou *shift right* (>>). Contudo, a mesma função pode ser obtida através da multiplicação por 2 (*shift left*) ou divisão por 2 (*shift right*).

Todas as operações matemáticas são obrigatoriamente realizadas da esquerda para a direita. Não é possível utilizar parêntesis. Por exemplo

```
let w1 = w2 / ( 2 + b3 ) não é uma expressão válida.
```

Deve ser substituída por

```
let b3 = 2 + b3
let w1 = w2 / b3
```

As instruções adição (+) e a subtracção (-) funcionam no modo normal. Note que as variáveis transbordam (*overflow*) sem aviso se o máximo ou o mínimo valores excederem o intervalo 0-255 para variáveis tipo byte ou 0-65535 para variáveis tipo word.

Na multiplicação de dois números de 16 bits, o resultado possui até 32 bits. A operação multiplicação (*) retorna a word mais baixo do resultado. A operação ** retorna a word mais alta do resultado.

A operação divisão (/) retorna o quociente word de uma divisão word/word. A operação módulo (// ou %) retorna o resto da divisão.

A operação MAX é um factor limitador, que assegura que o valor nunca excede um determinado valor. Neste exemplo o valor nunca excede 50. Quando o resultado da multiplicação exceder 50, a operação *max* limita o valor a 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 = 60      ' limitado a 50

```

Os operadores lógicos AND, OR, XOR, NAND, NOR e XNOR funcionam sobre as variáveis bit a bit.

Uma aplicação comum do operador AND (&) é a mascarar bits individuais:

```
let b1 = pins & %00000110
```

Neste caso, as entradas 1 e 2 são mascaradas, pelo que b1 apenas contém informação sobre elas.

Exemplo:

```

início:
  let b0 = b0 +1          ' incrementa b0
  sound 7, (b0, 50)      ' produz um som de tom b0 e duração 50
  if b0 > 50 then rest   ' depois de 50 salta para rest
  goto início           ' salta para início

rest:
  let b0 = b0 max 10     ' faz o resto de b0 para 10
                        ' pois 10 é o valor máximo
  goto início           ' salta para início

```

let dirs =

let dirsc =

Sintaxe:

{LET} dirsc = valor

- Valor(es) é uma variável/constante que actua sobre o registo que define o sentido dos pinos (entrada ou saída)

Função:

Configura os pinos do portc como entradas ou saídas.

Informação:

Alguns microcontroladores permitem que os pinos sejam configurados como entradas ou saídas. Nestes casos é necessário informar o microcontrolador quais os pinos a usar como entradas e/ou como saídas (são todos configurados como entradas quando o microcontrolador é alimentado). Existem dois modos de o fazer:

- 1) Utilizar uma instrução de saída (*high, pulsout, etc.*) que configura automaticamente o pino como uma saída.
- 2) Utilizar a declaração *let dirs = byte* de definição.

Neste último caso convencionou-se o uso da notação binária. Assim, o pino 7 é o bit mais à esquerda e o pino 0 o mais à direita. Se o bit for colocado a 0 o pino é uma entrada, se o pino for colocado a 1 será uma saída.

let pins =

let pinsc =

Sintaxe:

```
{LET} pins = valor  
{LET} pinsc = valor
```

- Valor(es) são variáveis/constantes que actuam sobre o porto de saída

Função:

Liga/desliga todas as saídas do porto principal de saída (let pins =).

Liga/desliga todas as saídas do portc (let pinsc =)

Informação:

As instruções *low* e *high* podem ser usadas para ligar individualmente os pinos como baixos ou altos.

Contudo, quando se trabalha com muitas saídas é por vezes conveniente alterá-las todas simultaneamente. Convencionou-se usar a notação binária quando se usa esta instrução. Assim, o pino 7 é o bit mais à esquerda e o pino 0 o mais à direita. Se o bit for colocado a 0 a saída ficará baixa (desligada), se o bit for colocado a 1 será uma saída alta (ligada).

Note que em dispositivos que possuem pinos bidireccionais, este comando apenas funciona em pinos de saída. Neste caso, é preciso configurar os pinos como saídas (utilizando uma instrução *let dirs =*) antes de usar esta instrução.

Exemplo:

```
let pins = %11000011      ‘ liga os pinos de saída 7,6,1,0.  
pause 1000                ‘ espera 1 segundo  
let pins = %00000000      ‘ desliga as saídas
```

lookdown

Sintaxe:

LOOKDOWN alvo, (valor1, valor2, ...,valorN), variável

- Variável recebe o resultado (se existir).
- Alvo é uma variável/constante que vai ser compara com os valores.
- Valores são variáveis/constantes.

Função:

Atribui à variável o valor da lista correspondente à posição alvo, caso exista.

Informação:

A instrução *lookdown* deve ser usada quando se possui um valor específico com o qual comparar uma tabela de valores. A variável alvo é comparada com os valores entre parêntesis. Se coincidir com o 5º item (valor4) a instrução retorna o número 4 na variável. Note que os valores são numerados a partir de 0 e não de 1. Se não existir coincidência o valor da variável fica inalterado.

Exemplo:

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

Neste exemplo a variável b2 irá conter o valor 3 se b1 contiver "d" e o valor 4 se b1 contiver "e".

lookup

Sintaxe:

LOOKUP offset, (dado0, dado1, ..., dadoN), variável

- Variável recebe o resultado (se houver)
- Offset é uma variável/constante que especifica qual o dado #(0-N) a colocar na variável.
- Dado0-N são variáveis/constantes.

Função:

Procura o dado especificado pelo *offset* na tabela e guarda-o na variável (se dentro da gama).

Informação:

A instrução *lookup* é utilizada para carregar uma variável com diferentes valores. O valor a ser carregado é o da posição da tabela definida pelo offset. No exemplo abaixo, se $b0 = 0$ então $b1$ será igual a “a”, se $b0 = 1$ então $b1$ será igual a “b”, etc. Se o *offset* exceder o número de entradas da tabela, o valor da variável fica inalterado.

Exemplo:

início:

```
let b0 = b0 +1           ‘ incrementa b0
lookup b0, (“1234”), b1 ‘ coloca carácter ASCII em b1
if b0 < 4 then início   ‘ ciclo
end
```

low

Sintaxe:

LOW pino

- Pino é uma variável/constante (0-7) que especifica o pino de E/S a usar.

Função:

Coloca o pino no nível lógico baixo.

Informação:

A instrução *low* desliga uma saída, colocando-a no nível lógico baixo.

Exemplo:

```
ciclo:
high 1           ‘ liga a saída 1
pause 500       ‘ espera 0,5 segundos
low 1          ‘ desliga a saída 1
pause 500       ‘ espera 0,5 segundos
goto ciclo
```

nap

Sintaxe:

NAP período

- Período é uma variável/constante que determina a duração da suspensão de funcionamento a consumo reduzido. A duração é calculada como $2^{\text{período}} * 18 \text{ ms}$ (aproximadamente). O período pode variar entre 0 e 7.

Função:

Suspende a operação do microcontrolador durante o período indicado. O consumo é reduzido em função do valor do período.

Informação:

A instrução *nap* coloca o microcontrolador no modo de consumo reduzido durante um curto período de tempo. Quando está no modo de consumo reduzido, todos os temporizadores são desligados, pelo que as instruções *servo* e *pwmout* deixam de funcionar. O período de tempo nominal é dado pela tabela junta. Devido a tolerâncias nos temporizadores, a duração está sujeita a -50 a +100% de tolerância. A temperatura ambiente também afecta esta tolerância, pelo que em nenhum projecto que necessite de uma base de tempo precisa se deve usar esta instrução.

Período	Tempo de atraso
0	18 ms
1	36 ms
2	72 ms
3	144 ms
4	288 ms
5	576 ms
6	1,152 s
7	2,304 s

A instrução *nap* usa o temporizador do *watchdog* interno, que não é alterado por alteração na frequência do relógio do sistema.

Exemplo:

```
ciclo:
  high 1      ' liga a saída 1
  nap 4      ' suspende durante 2^4 x 18ms
  low 1      ' desliga a saída 1
  nap 7      ' suspende durante 2^7 x 18ms
  goto ciclo ' salta para o início
```

on...goto

Sintaxe:

ON offset GOTO endereço0, endereço1 ... endereçoN

- Offset é uma variável/constante que especifica qual o Endereço # a usar (0-N).
- Endereços são *labels* que especificam para onde saltar.

Função:

Desvia o programa para o endereço especificado pelo *offset* (se pertencer ao intervalo).

Informação:

Esta instrução permite saltar para diferentes posições dependendo do valor da variável 'offset'. Se o *offset* tiver o valor 0, o programa continua no endereço0, se o *offset* tiver o valor 1 o programa continua no endereço1, etc. Se o *offset* for superior ao número de endereços a instrução é ignorada e o programa continua na linha seguinte.

Esta instrução é idêntica à instrução *branch*.

Exemplo:

```
reset:  let b1 = 0
        low 0
        low 1
        low 2
        low 3
princ:  let b1 = b1 + 1
        if b1 > 3 then reset
        on b1 goto btn0,btn1, btn2, btn3
        goto princ
btn0:   high 0
        goto princ
btn1:   high 1
        goto princ
btn2:   high 2
        goto princ
btn3:   high 3
        goto princ
```

on...gosub

Sintaxe:

ON offset GOSUB endereço0, endereço1 ... endereçoN

- Offset é uma variável/constante que especifica qual a subrotina a usar (0-N).
- Endereços são *labels* que especificam a localização da subrotina a usar pela instrução *gosub*.

Função:

Executa um *gosub* para o endereço especificado pelo *offset* (se contido no intervalo).

Informação:

Esta instrução possibilita a chamada condicional de uma subrotina em função do valor contido na variável ‘offset’. Se o *offset* contiver o valor 0, o programa chama a subrotina através de um *gosub* para o endereço0, se o *offset* contiver o valor 1 será chamada a subrotina de endereço1, etc.

Se o *offset* for superior ao número de endereços a instrução é ignorada e o programa continua na linha seguinte.

A instrução *return* contida na subrotina faz o programa regressar à instrução seguinte à linha onde se deu a chamada (*on...gosub*).

Exemplo:

```
reset: let b1 = 0
        low 0
        low 1
        low 2
        low 3
princ:  let b1 = b1 + 1
        if b1 > 3 then reset
        on b1 gosub btn0,btn1, btn2, btn3
        goto princ
btn0:  high 0
        return
btn1:  high 1
        return
btn2:  high 2
        return
btn3:  high 3
        return
```

pause

Sintaxe:

PAUSE milisegundos

- Milisegundos é uma variável/constante (0-65535) que especifica quantos milisegundos dura a pausa.

Função:

Pausa com a duração especificada em milisegundos. A duração da pausa na execução do programa possui a precisão do relógio do microcontrolador – neste caso um ressoador cerâmico a 4 MHz.

Informação:

A instrução *pause* cria um atraso de tempo (em milisegundos a 4 MHz). O maior atraso possível é de cerca de 65 segundos. Para criar um atraso de maior duração (por ex. 5 minutos) deverá utilizar um ciclo *for...next*.

```
for b1 = 1 to 5           ' 5 ciclos
    pause 60000          ' espera de 60 segundos
next b1
```

Durante a pausa o único modo de reagir às entradas é através de *interrupts* (veja a informação sobre a instrução *setint*). Não inclua longas pausas em ciclos que se destinam a detectar alterações nas entradas.

Exemplo:

```
ciclo:
    high 1                ' liga a saída 1
    pause 500             ' espera 0,5 segundos
    low 1                  ' desliga a saída 1
    pause 500             ' espera 0,5 segundos
goto ciclo
```

pulsin

Sintaxe:

PULSIN pino, estado, variável

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.
- Estado é uma variável/constante (0 ou 1) que especifica que flanco deve ocorrer para se iniciar a medição do impulso – contagem de unidades de 10us (para um relógio de 4 MHz).
- Variável recebe o resultado (1-65536). Se ocorrer timeout (0,65536 s) o resultado será 0.

Função:

Mede a largura de um impulso no pino e guarda o resultado na variável.

Informação:

A instrução *pulsin* mede a largura de um impulso. Se não ocorrer nenhum impulso, o resultado será 0. Se estado = 1 então o início da contagem inicia-se numa transição do sinal de baixo para alto. Se estado = 0 então o início da contagem inicia-se numa transição do sinal de alto para baixo. Utilize a instrução *count* para contar o número de impulsos num determinado período.

Exemplo:

```
pulsin 3, 1, w1          ' guarda a duração do impulso no pino 3 em w1
```

pulsout

Sintaxe:

PULSOUT pino, tempo

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.
- Tempo é uma variável/constante que especifica o período (0-65535) em unidades de 10us (para um relógio de 4 MHz).

Função:

Produz um impulso de duração especificada por tempo, invertendo o valor lógico do pino.

Informação:

A instrução *pulsout* gera um impulso de uma dada duração. Se a saída estiver inicialmente baixa, o impulso será alto, e vice-versa. Esta instrução configura automaticamente os pinos como saídas.

Exemplo:

```
ciclo:
    pulsout 4, 150      ' envia um impulso de 1,5 ms para o pino 4
    pause 20           ' pausa de 20 ms
    goto ciclo         ' salto para o início
```

pwmout

Sintaxe:

PWMOUT pino, período, dutycycle

- Pino é uma variável/constante que especifica o pino E/S a usar (1 ou 2).
- Período é uma variável/constante (0-255) que estabelece o período do sinal de PWM.
- Duty cycle é uma variável/constante (valor de 10 bits, logo, de 0-1024) que define o ciclo de trabalho (tempo em que o sinal está alto em cada período).

Função:

Gera um sinal contínuo *pwm* usando o módulo interno do microcontrolador.

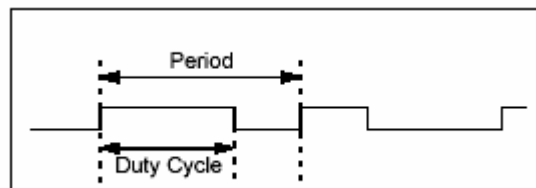
Informação:

Esta instrução **difere de todas as outras** pelo facto de ser executada continuamente (independente do resto do programa) até que outra instrução *pwmout* seja enviada. Para parar o sinal *pwmout*, basta enviar uma instrução *pwmout* com o período 0.

Período PWM = (período + 1) x 4 x (1/4000000) = (período + 1) us

Frequência PWM = 1/ (período PWM)

Ciclo Trabalho = (duty cycle) x (1/4000000) = (duty cycle)/4 us



Como esta instrução utiliza o módulo pwm do microcontrolador existem restrições a observar:

- 1) O módulo *pwm* utiliza o mesmo temporizador para os dois pinos *pwmout*. Desse modo, quando se usam as duas saídas *pwm*, o período é comum.
- 2) A instrução *servo* não pode ser executada ao mesmo tempo que as instruções *pwmout* pois utilizam o mesmo temporizador.
- 3) O sinal *pwmout* pára durante a execução da instrução *sleep* ou após uma instrução *end*.
- 4) Das equações acima resulta que o valor máximo do ciclo de trabalho não pode ultrapassar 4 x período. O uso de valores superiores produz resultados imprevisíveis.

Exemplo:

motor:

```
pwmout 2, 150, 100    ‘ estabelece o pwm com período de 151us
                        ‘ com 25us alto e o restante baixo.
pause 1000             ‘ pausa de 1 segundo
goto motor             ‘ salto para início
```

random

Sintaxe:

RANDOM variável

- Variável é simultaneamente o espaço de trabalho e o resultado. Como a instrução `random` gera uma sequência pseudo-aleatória é aconselhável chamá-lo repetidamente dentro de um ciclo.

Função:

Gera um número pseudo-aleatório na variável.

Informação:

A instrução `random` gera uma sequência de números pseudo-aleatórios entre 0 e 65535. Todos os microcontroladores precisam de realizar operações matemáticas para gerar números aleatórios, pelo que a sequência nunca é realmente aleatória. Nos computadores utiliza-se algo que muda (como a hora do sistema) para iniciar o cálculo, pelo que cada instrução `random` produz resultados diferentes. Os Picaxe não possuem essa funcionalidade, pelo que a sequência gerada será sempre idêntica, a não ser que o valor da variável seja modificado antes de cada instrução `random`. O modo mais simples de realizar uma sequência aleatória consiste em repetidamente executar a instrução `random` num ciclo, enquanto espera pela pressão num botão. Como o número de ciclos varia entre duas pressões no botão, a saída é bastante mais aleatória.

Embora apenas precise de uma variável `byte` (0-255), utilize na mesma uma variável `word` (por ex: `w0`) na instrução. Como `w0` é constituída por `b0` e `b1`, pode usar qualquer um dos `bytes` como número aleatório.

Exemplo:

```
ciclo:
    random w0          ' coloca um número aleatório em w0
    if pin1 = 1 then fazer
    goto ciclo

fazer:
    let pins = b1      ' coloca o número aleatório nos pinos de saída
    pause 100         ' espera de 0,1 s
    goto ciclo        ' salto para o início
```

readadc

Sintaxe:

READADC canal, variávelbyte

- Canal é uma variável/constante especificando um endereço (0-3).
- Variávelbyte é uma variável byte (0-255) que recebe os dados lidos.

Função:

Lê um canal ADC (conversão analógico-digital) de 8 bits de resolução. Para uma variável.

Informação:

A instrução *readadc* é utilizada para ler um valor analógico de um pino do microcontrolador. Note que nem todos os pinos possuem esta funcionalidade. No Picaxe-28X apenas podem ser utilizados os pinos 0, 1, 2 e 3. O valor calculado pelo conversor A/D possui 10 bits de resolução mas é arredondado para 8 bits. Deve usar-se a instrução *readadc10* para obter um valor de 10 bits.

Exemplo:

início:

```
readadc 1, b1      ‘ lê tensão analógica presente no canal 1 e converte-o para
                   ‘ valor de 8 bits em b1
if b1 > 50 then flsh ‘ salta para a subrotina flsh se b1 for maior que 50
goto início       ‘ caso contrário salta para o início e repete
```

flsh:

```
high 1            ‘ liga a saída 1
pause 5000        ‘ espera 5 segundos
low 1             ‘ desliga a saída 1
pause 5000        ‘ espera 5 segundos
goto início       ‘ salto para o início
```

readadc10

Sintaxe:

READADC10 canal, variávelword

- Canal é uma variável/constante especificando um endereço (0-3).
- Variávelword é uma variável word que recebe os dados lidos (0-1024).

Função:

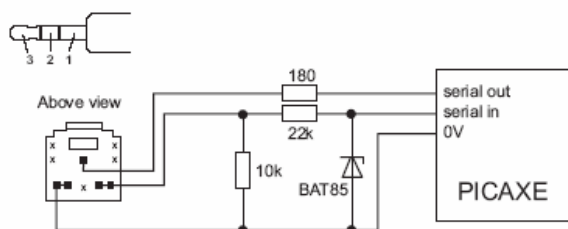
Lê um canal ADC (conversão analógico-digital) de 10 bits de resolução para uma variável de dimensão word (0-1024).

Informação:

A instrução *readadc10* é utilizada para ler um valor analógico com 10 bits de resolução. Note que nem todos os pinos possuem esta funcionalidade. No Picaxe-28X apenas podem ser utilizados os pinos 0, 1, 2 e 3.

Como o resultado possui 10 bits é necessária uma variável *word*.

Quando se usa a instrução *debug* para visualizar dados de 10 bits, a ligação ao computador através do cabo de *download* pode afectar ligeiramente os resultados da conversão. Neste caso, recomenda-se uma melhoria no cabo, como se mostra na figura.



Exemplo:

início:

```
readadc10 1, w1    ' lê tensão analógica presente no canal 1 e converte-o para w1
debug w1           ' transmite ao computador
pause 200          ' espera curta
goto início        ' salto para o início
```

read

Sintaxe:

READ localização, variável

- Localização é uma variável/constante especificando um endereço (0-127)
- Variável recebe o dado lido.

Função:

Lê o conteúdo de uma localização na *eeeprom* e guarda-o na variável.

Informação:

A instrução *read* permite a leitura de dados byte da memória de dados do microcontrolador. O conteúdo desta memória não se perde quando se desliga a alimentação. Contudo, os dados são actualizados (através da instrução EEPROM) sempre que se faz *download* do programa. Para guardar dados em memória no decorrer de um programa use a instrução *write*.

A instrução *read* é orientada ao byte, pelo que a leitura de uma variável *word* é feita por duas instruções, uma leitura para cada *byte*. O PICAXE 28X possui 128 posições de memória na *eprom*.

Exemplo:

main:

```
for b0 = 0 to 63          ' inicia o ciclo
  read b0, b1            ' lê valor da EEPROM e guarda na var b1
  serout 7, T2400, (b1)  ' transmite em série para o display LCD
next b0                  ' próximo dado
```

return

Sintaxe:

RETURN

Função:

Retorno de uma subrotina (procedimento).

Informação:

A instrução *return* apenas se usa com a respectiva instrução *gosub*, para retornar do fim de uma subrotina ao programa principal. O uso de *return* sem o correspondente *gosub* produz uma falha na execução do programa.

Exemplo:

ciclo:

```
let b2 = 15              ' atribui 15 ao valor de b2
pause 2000               ' espera 2 segundos (2000 milisegundos)
gosub flsh               ' chama procedimento flsh
let b2 = 5               ' atribui 5 ao valor de b2
pause 2000               ' espera 2 segundos (2000 milisegundos)
gosub flsh               ' chama procedimento flsh
end                       ' impede entrada acidental no procedimento
```

flsh:

```
for b0 = 1 to b2         ' define b2 como número de ciclos
  high 1                 ' liga a saída 1
  pause 500              ' espera 0,5 segundos
  low 1                  ' desliga a saída 1
  pause 500              ' espera 0,5 segundos
next b0                  ' fim do ciclo
return                   ' retorno do procedimento
```

select case \ case \ else \ endselect

Sintaxe:

```
SELECT VAR  
CASE VALOR  
  {codigo}  
CASE VALOR, VALOR...  
  {codigo}  
CASE VALOR TO VALOR  
  {codigo}  
CASE ?? valor  
  {codigo}  
ELSE  
  {codigo}  
ENDSELECT
```

- Var é o valor a testar.
- Valor é a variável/constante.

?? pode ser qualquer uma das seguintes condições

=	igual a
<>	não igual a (diferente)
!=	não igual a (diferente)
>	maior que
>=	maior que ou igual a
<	menor que
<=	menor que ou igual a

Função:

Compara o valor da variável e executa condicionalmente secções de código.

Informação:

A instrução múltipla *select \ case \ else \ endselect* é utilizada para testar uma variável a determinadas condições. Se a condição é verificada então essa secção de código de programa é executada, e o programa salta para a posição *endselect*. Se a condição não é verificada o programa segue na instrução *case* ou *else* seguinte. A secção de código *else* só é executada se nenhuma das condições se verificar.

Exemplo:

```
select case b1  
  case 1  
    high 1  
  case 2,3  
    low 1  
  case 4 to 6  
    high 2  
  else  
    low 2  
endselect
```

serin

Sintaxe:

SERIN pino, baudmode, (qualificador, qualificador,...)

SERIN pino, baudmode, (qualificador, qualificador,...), {#}variável, ...

SERIN pino, baudmode, {#}variável, {#}variável...

Função:

Entrada série com qualificadores optativos (8 dados, no parity, 1 stop).

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.
- *baudmode* é uma variável/constante que especifica o modo:

T2400	true input	(todas as <i>baudrates</i> referidas a 4 MHz)
T1200	true input	
T600	true input	
T300/T4800	true input	
N2400	inverted input	
N1200	inverted input	
N600	inverted input	
N300/N4800	inverted input	

- Qualificadores são variáveis/constants opcionais (0-255) que devem ser recebidas na ordem correcta antes que quaiqueer outros bytes susequentes possam ser recebidos e armazenados nas variáveis.
- Variável(s) recebe o resultado(s) (0-255). Os #s opcionais destinam-se à entrada de números decimais ASCII em variáveis, em vez de caracteres.

Função:

Entrada série com qualificadores opcionais (8 bits dados, sem paridade, 1 bit stop).

Informação:

A instrução *serin* é utilizada para receber dados em série num pino de entrada do microcontrolador. Não pode ser usada no pino de entrada série de *download*, que está reservado apenas para esse efeito.

Pino especifica o pino de entrada a ser utilizado. *Baude mode* especifica a velocidade de transferência e a polaridade do sinal. No caso de se usar a interface simples com resistências, deve usar-se sinais N (invertidos). Quando se utilizar uma interface com MAX232 deve utilizar-se sinais T (true). O protocolo é fixado para N, 8, 1 (no parity, 8 bits de dados e 1 stop bit).

Note que o microcontrolador pode não ser capaz de lidar com datagramas complexos à velocidade máxima de 4800 baud – recomenda-se como máximo 2400 baud para um relógio de 4 MHz.

São usados qualificadores para especificar um *byte* “marcador” ou sequência. A instrução

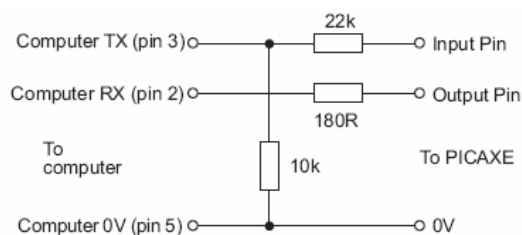
```
serin 1, N2400, (“ABC”), b1
```

exige a recepção da string “ABC” antes da leitura do próximo byte a colocar em b1.

Sem qualificadores

```
serin 1, N2400, b1
```

o primeiro *byte* recebido será colocado em b1. Todo o processamento pára até à recepção do novo *byte* de dados série. Esta instrução não pode ser interrompida pela instrução *setint*.



Exemplo de circuito de interface com o computador

serout

Sintaxe:

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

- Pino é uma variável/constante (0-7) que especifica o pino de E/S a usar.
- Baudmode é uma variável/constante (0-7) que especifica o modo:

T2400	true ouput	(todos os baudrates referidos a 4 MHz)
T1200	true ouput	
T600	true ouput	
T300/T4800	true ouput	
N2400	inverted ouput	
N1200	inverted ouput	
N600	inverted ouput	
N300/N4800	inverted ouput	

- Data são variáveis/constantes (0-255) que fornecem os dados a serem enviados.
- Os caracteres #s opcionais destinam-se ao envio de números decimais em caracteres ASCII, em vez do dado bruto. O texto pode ser colocado entre aspas (“Hello”).

Função:

Transmite dados em série por um pino de saída (8 data bits, no parity, 1 stop bit).

Informação:

A instrução *serout* é usada para transmitir dados em série por um pino de saída do microcontrolador. Não pode ser usada com a saída série para *download* de programas – para isso, deve usar a instrução *sertxd*.

Pino especifica o pino de saída a usar. *Baud mode* especifica a velocidade de transmissão (*baud rate*) e a polaridade do sinal. Quando utilizar a interface simples de resistências, seleccione N (sinal invertido). Se utilizar uma interface com o integrado MAX232 utilize sinais T (sinal directo). O protocolo é fixo para N, 8, 1 (no parity, 8 data bits, 1 stop bit).

Note que a 4800 bauds o microcontrolador pode não ser capaz de lidar com datagramas complexos (recomenda-se 2400 para um relógio de 4 MHz).

O símbolo # possibilita o envio de caracteres *ascii*. Assim, #b1, se b1 contiver o dado 126, irá enviar os caracteres *ascii* “1” “2” “6” em vez do dado 126.

Veja ainda as notas à instrução *serin*.

Exemplo:

```
ciclo:
  for b0 = 0 to 63
    read b0, b1
    serout 7, N2400, (b1)
  next b0
```

‘ inicia o ciclo
‘ lê o valor em b1
‘ transmite o valor
‘ ciclo seguinte

sertxt

Sintaxe:

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

- Data são variáveis/contantes (0-255) que fornecem dados para serem enviados para a saída.

Função:

Saída série através do pino de download (4800 baud, 8 data bits, no parity, 1 stop bit).

Informação:

A instrução *sertxd* é semelhante à instrução *serout*, mas diz respeito ao pino de saída série e não a um pino genérico de saída qualquer. Isso permite enviar dados de volta ao computador através do cabo de programação. A instrução pode ser muito útil durante a fase de testes – visualização dos dados na janela PICAXE>Terminal window. Existe uma opção em View>Options para abrir automaticamente a janela Terminal a seguir a um download.

A velocidade de transferência (baud rate) está fixa em 4800, n, 8, 1.

Exemplo:

```
ciclo:
    for b1 = 0 to 63
        sertxd ("Valor de b1 ", #b1,13,10)
        pause 1000
    next b1
```

‘ início do ciclo
‘ novo ciclo

servo

Sintaxe:

SERVO pino, impulso

- Pino é uma variável/constante (0-7) que especifica o pino E/S a utilizar.
- Impulso é a variável/constante (75-225) que especifica a posição do servo.

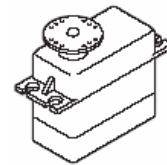
Função:

Envia periodicamente um impulso para o pino de saída para controlar um servo.

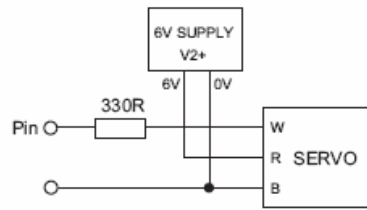
Informação:

Os servos normalmente encontrados em brinquedos rádio controlados, são grupos motor/redutor muito precisos que podem ser movidos repetidamente para uma mesma posição, através de um sensor de posição interno. Normalmente os servos requerem um impulso de 0,75 a 2,25ms em cada 20ms, e este impulso tem que ser enviado periodicamente cada 20ms. Uma vez perdido o impulso o servo perde a sua posição.

A instrução *servo* inicia o envio de impulsos por um pino colocando-o alto o durante um intervalo de tempo que é o dado impulso x 0,01ms, em cada 20ms. Esta instrução é diferente de todas as outras (excepto *pwmout*) pelo facto de os impulsos se manterem continuamente até que nova instrução *servo* altere as suas especificações. Uma instrução *high* ou *low* no pino, interrompe os impulsos imediatamente. As instruções *servo* ajustam a largura do impulso ao novo valor, movendo o servo para a nova posição. A instrução *servo* não pode ser usada ao mesmo tempo que a instrução *pwmout*, pois usam o mesmo temporizador interno.



Não se deve utilizar valores de impulso abaixo de 75 nem acima de 225, pois isso pode provocar avarias nos servos. Devido a tolerâncias de fabrico os valores indicados são aproximados e devem ser ajustados caso a caso.



Deve utilizar sempre uma alimentação separada de 6 V (4 x 1,5 V) para os servos pois este geram muito ruído eléctrico.

Note ainda que o tempo necessário para executar a instrução *servo* todos os 20 ms leva a que todas as outras levem mais tempo a executar, nomeadamente a instrução *pause* que levará ligeiramente mais tempo do que o devido. Os impulsos servo são temporariamente desligados durante as instruções sensíveis ao tempo como *serin*, *serout*, *sertxd* e *debug*.

Exemplo:

ciclo:

```

servo 4, 75      ‘ move o servo para uma das posições finais
pause 2000      ‘ espera 2 segundos
servo 4, 150    ‘ move o servo para a posição central
pause 2000      ‘ espera 2 segundos
servo 4, 225    ‘ move o servo para a posição final oposta
pause 2000
goto ciclo      ‘ volta para ciclo
  
```

setint

Sintaxe:

SETINT entrada, máscara

- Entrada é uma variável/constante (0-255) que especifica as condições de entrada.
- Máscara é uma variável/constante (0-255) que especifica a máscara.

Função:

Produz uma interrupção (interrupt) numa determinada condição.

Informação:

A instrução *setint* produz uma interrupção por sondagem perante a realização de uma dada condição num dado pino de entrada.

Uma interrupção por sondagem (*polled interrupt*) é uma maneira rápida de reagir a uma determinada combinação de entradas. Trata-se do único tipo de *interrupt* disponível no Picaxe. O porto de entradas é verificado após a execução de cada uma das instruções do programa e continuamente durante a execução de uma instrução

pause. Se a condição for verdadeira, é executada uma instrução *gosub* para o procedimento *interrupt* que é executado imediatamente. Quando o procedimento tiver sido executado, a execução continua no programa principal a partir da instrução seguinte aquela onde se deu a interrupção.

A condição de entradas da *interrupção* é qualquer padrão de 0s e 1s do porto de entrada, mascarada pelo *byte* da máscara. Portanto, todos os bits mascarados com 0 na máscara serão ignorados.

Por exemplo:

Para haver *interrupção* quando a entrada 1 (input1) estiver alta
setint %00000010, %00000010

Para haver *interrupção* t quando a entrada 1 (input1) estiver baixa
setint %00000000, %00000010

Para haver *interrupção* quando input0 alto, input1 alto e input2 baixo
setint %00000011, %00000111

etc.

Só é permitido um padrão de entradas em cada momento. Para desactivar a *interrupção* basta executar a instrução *setint* com o valor 0 como máscara.

Notas:

- 1) Cada programa que usa a instrução *setint* deve possuir a correspondente procedimento *interrupt* (subrotina terminada com a instrução *return*), no fim do programa.
- 2) Quando ocorre a *interrupção*, esta fica inibida. Portanto, para voltar a habilitar a *interrupção* é necessário colocar uma instrução *setint* no procedimento *interrupt*. A *interrupção* não é habilitada senão após a execução da instrução *return*.
- 3) Se a *interrupção* é re-habilitada e a condição de interrupção se mantiver, verificar-se-á nova *interrupção* após a instrução *return*.
- 4) Após a execução do código do procedimento, a execução continua no programa principal a partir da instrução seguinte àquela onde se deu a *interrupção*. No caso de ter sido uma instrução *pause* ou *wait*, o tempo restante do atraso é ignorado e o programa continua na linha seguinte.

Para mais informação veja o manual original.

Exemplo:

```
setint %10000000, %10000000      ‘ activa a interrupção quando o pino 7 ficar alto
```

```
ciclo:
    low 1                          ‘ desliga (off) a saída 1
    pause 2000                      ‘ espera 2 segundos
    goto ciclo                       ‘ volta ao início
```

```
interrupt:
    high 1                          ‘ liga (on) a saída 1
```

```

if pin7 = 1 then interrupt      ‘ permanece aqui até a condição de interrupção ser limpa

pause 2000                    ‘ espera 2 segundos
setint %10000000, %10000000  ‘ reactiva a interrupção
return                        ‘ retorno da subrotina

```

sleep

Sintaxe:

SLEEP período

- Período é uma variável/constante (0-65535) que especifica a duração da instrução em múltiplos de 2,3 segundos.

Função:

Fica em estado *sleep* durante o período especificado (período x 2,3 s).

Informação:

A instrução *sleep* coloca o microcontrolador no modo de baixo consumo durante um certo tempo. Neste estado de baixo consumo, os temporizadores internos são desligados, pelo que as instruções *pwmout* e *servo* deixam de funcionar. O período nominal é de 2,3 s, pelo que um *sleep 10* corresponde a 23 segundos. A instrução *sleep* não é precisa, pelo que devido a tolerâncias de fabrico dos temporizadores internos, existe uma tolerância de -50% a +100%. A temperatura ambiente também influencia esta tolerância, pelo que não deve ser utilizada esta instrução como base de tempo, em programas que exijam precisão.

Para “sonos” mais curtos pode usar-se a instrução *nap*.

Exemplo:

```

ciclo:
high 1          ‘ liga (on) a saída 1
sleep 10      ‘ dorme durante 23 segundos
low 1          ‘ desliga (off) a saída 1
sleep 100    ‘ dorme durante 230 segundos
goto ciclo     ‘ volta ao início

```

sound

Sintaxe:

SOUND pino, (nota, duração, nota, duração, ...)

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.

- Nota(s) são variáveis/constantes (0-255) que especificam o tipo e a frequência.
Nota 0 é silêncio. Notas 1-127 são tons crescentes. Notas 128-255 são ruídos brancos crescentes.
- Duração(s) são variáveis/constantes (0-255) que especificam a duração múltiplos aproximados de 10ms).

Função:

Produz sons e ruídos.

Informação:

Esta instrução é destinada a produzir sons audíveis (*bips*) para jogos e teclado. Nota e duração devem ser usados na instrução aos pares.

Exemplo:

ciclo:	let b0 = b0 + 1	‘ incrementa b0
	sound 7, (b0, 50)	‘ produz tom b0 de duração 0,5s no pino 7
	goto ciclo	‘ vai para ciclo

stop

Sintaxe:

STOP

Função:

Entra num ciclo permanente de paragem até que haja uma nova execução do programa ou o PC ligue para novo *download*.

Informação:

A instrução *stop* põe o microcontrolador num ciclo permanente no fim do programa. Ao contrário da instrução *end*, a instrução *stop* não põe o microcontrolador num estado de consumo reduzido, depois do programa terminar.

A instrução *stop* não desliga os temporizadores internos, pelo que as instruções *pwmout* e *servo* continuam a funcionar.

Exemplo:

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

symbol

Sintaxe:

SYMBOL nomesimbolo = valor
SYMBOL nomesimbolo = valor ?? constante

- Nomesimbolo é uma *cadeia de caracteres (string)* que deve começar com um carácter alfabético ou um “_”. Após o primeiro carácter, pode conter também caracteres numéricos (“0” – “9”).
- Valor é uma variável ou constante a que se atribui um nome simbólico alternativo.
- ?? pode ser qualquer uma das funções matemáticas suportadas.

Função:

Atribui um valor a um novo nome simbólico.

Os operadores matemáticos podem também ser usados em constantes (não variáveis).

Informação:

Os símbolos são utilizados para renomear constantes ou variáveis para torná-las mais fáceis de lembrar na programação. Os símbolos não produzem qualquer alteração ao tamanho dos programas pois são convertidos para “números” antes do *download*.

Os nomes dos símbolos podem conter quaisquer caracteres, mas não podem começar com um carácter numérico. Naturalmente os nomes dos símbolos não podem ser palavras reservadas como *input*, *step*, *pause*, etc.

Quando utilizar definições de pinos de entrada ou saída tenha cuidado em usar o termo “pin0” e não “0” para descrever as variáveis de entrada nas instruções *if...then*.

Exemplo:

```
symbol RED_LED = 7           ‘ define um pino de saída
symbol CONTA = B0           ‘ define o símbolo de uma variável
let CONTA = 200             ‘ carrega a variável com o valor 200
CICLO:                      ‘ define endereço de programa:
    high RED_LED           ‘ um símbolo endereço termina por dois pontos
    pause CONTA           ‘ liga a saída 7
                          ‘ espera 0,2 segundos (200 milisegundos)
```

```
low RED_LED
pause CONTA
goto CICLO
```

```
‘ desliga a saída 7
‘ espera 0,2 segundos (200 milisegundos)
‘ efectua um salto para o início CICLO (ciclo)
```

toggle

Sintaxe:

TOGGLE pino

- Pino é uma variável/constante (0-7) que especifica o pino E/S a usar.

Função:

Configura o pino como saída e muda-lhe o estado (de 1 para 0, ou de 0 para 1)

Informação:

A instrução *toggle* inverte uma saída (se estiver baixa passa a alta e vice-versa).

Exemplo:

```
início:
toggle 7      ‘ faz toggle à saída 7
pause 1000    ‘ espera de 1 segundo
goto início   ‘ volta para início
```

write

Sintaxe:

WRITE localização, dado

- Localização é uma variável/constante que especifica o endereço de um byte (0-255).
- Dado é uma variável/constante que fornece o dado a ser escrito em memória.

Função:

Escreve o conteúdo de um dado de um byte na memória de dados.

Informação:

A instrução *write* possibilita a escrita de dados de um *byte* na memória de dados. O conteúdo da memória não se perde quando existe corte da alimentação. Contudo, os dados são actualizados (através da instrução EEPROM) quando há um novo download. Para ler os dados durante a execução do programa utilize a instrução *read*.

A instrução *read* é orientada ao byte, pelo que a leitura de uma variável *word* é feita por duas instruções, uma leitura para cada *byte*. O PICAXE 28X possui 128 posições de memória na *eeprom*.

Exemplo:

```
ciclo:
  for b0 = 0 to 63
    serin 6, T2400, b1
    write b0, b1
  next b0
```

‘ início do ciclo
‘ recebe um valor série
‘ escreve o valor em b1
‘ ciclo seguinte