PIC LOGO

Language Reference Manual

Overview

PIC Logo is the programming environment for the PIC Tower and the PIC Logo Chips based on a PIC16F87x microcontroller. (See the PIC Tower and the PIC Logo Chip hardware overview document.) PIC Logo supports:

- the ability to directly write and read all microcontroller registers
- control structures like `if`, `repeat`, `wait`, `waituntil` and `loop`
- global and local variables
- procedure definition with inputs and return values
- a multitasking `when` primitive
- a 16-bit number system (addition, subtraction, multiplication, division, comparison);
- timing functions and a random number function

When using PIC Logo, user programs are entered on a desktop computer and compiled into tokens that are transferred to the Tower through the serial port of the host computer. Logo commands can be executed by typing a line in the "**command center**" under the PIC Logo tab in the Tower Development Environment and pressing the <ENTER> key.

PIC Logo is a procedural language; procedures are defined using Logo to and end syntax:

to <*procedure-name*>
<*procedure-body*>
end

User defined procedures are downloaded to a PIC Tower or Logo Chip by clicking the "*Select File*" under the **PIC Logo** tab of the **Tower Development environment,** choosing the text file where these procedures are defined, and then clicking on "*Compile & Download File*".

Procedures can also be brought in from another file with "include". For example, in the PIC Logo **test.pl** file, the procedures from the **basic.inc** file are included with the following line:

include logochip/include/basic.inc

Note that the full path name relative to the folder in which the TDE resides. Using several includes, procedures from multiple text files can be incorporated into one document.

CONTROL STRUCTURES

if

DESCRIPTION

Executes a block of code if a condition expression is non-zero.

USAGE FORMAT

```
if    (conditional-expression)
      [code-block]
```

The ***conditional-expression*** is the expression whose value determines whether the code block is evaluated and ***code-block*** is the block of code to be evaluated.

Note: This primitive only actually checks the low byte of the result of the conditional expression (See the second sample code below.)

SAMPLE CODE AND OUTPUT

```
if 42       [print-string "Hi! cr]
>Hi!

if $4000    [print-string "Hi! cr]
>

if 0        [print-string "Hi! Cr]
>
```

ifelse

DESCRIPTION

Executes one of two blocks of code, depending on an expression's results. If the expression is non-zero, the first block of code is evaluated. If not, the second block of code is executed.

USAGE FORMAT

ifelse ***(conditional-expression)***
[*code-block-ture*]
[*code-block-false*]

The ***conditional-expression*** is the expression whose value determines which code block is evaluated. The ***code-block-true*** is the block of code to be evaluated if the condition is true and the ***code-block-false*** is the block of code to be evaluated if the condition is false.

Note: As for the **if** primitive, this primitive only checks the low byte of its conditional expression.

SAMPLE CODE AND OUTPUT

```
ifelse (0 = 1) [print 1][print 0]
>0

ifelse (1 = 1) [print 1][print 0]
>1

ifelse ($4000) [print 1][print 0]
>0

ifelse (42)    [print 1][print 0]
>1

ifelse (0)     [print 1][print 0]
>0
```

loop

DESCRIPTION

Repeats a block of code forever.

USAGE FORMAT

```
loop   [code-block]
```

The ***code-block*** is the block of code to be repeated forever.

Note: The **loop** can be broken by a **stop**, **stop!**, or **output** primitive.

SAMPLE CODE AND OUTPUT

```
to mwait :msecs
resett
loop [
if (timer > :msecs) [stop]
]
end
```

output

DESCRIPTION

Exits the currently running procedure and returns a value.

USAGE FORMAT

```
output      value
```

The ***value*** is value to return.

SAMPLE CODE AND OUTPUT

```
to add-numbers :x :y
output :x + :y
end
```

repeat

DESCRIPTION

Repeats a block of code for a given number of times.

USAGE FORMAT

repeat *count* [*code-block*]

The ***code-block*** is the block of code to be repeated and *count* is the numbers of times to repeat.

SAMPLE CODE AND OUTPUT

```
repeat 2 + 2 [print-string "Hi! send 10]
>Hi!
>Hi!
>Hi!
>Hi!
```

stop!

DESCRIPTION

Halts the virtual machine and whatever program is running.

USAGE FORMAT

```
stop!
```

SAMPLE CODE AND OUTPUT

```
if btst 3 porta [stop!]
```

stop

DESCRIPTION

Immediately exits the currently running procedure.

USAGE FORMAT

```
stop
```

SAMPLE CODE AND OUTPUT

```
to mwait :msecs
resett
loop [
if (timer > :msecs) [stop]
]
end
```

wait

DESCRIPTION

Waits for a specified number of tenths of a second. The process started by the **when** primitive continues to be checked during the wait.

USAGE FORMAT

```
wait duration
```

The *duration* argument is the number of tenths of seconds to wait.

SAMPLE CODE AND OUTPUT

```
wait 10
```

waituntil

DESCRIPTION

Repeatedly checks a condition until it becomes true, at which point it continues with the subsequent commands in the program.

USAGE FORMAT

```
waituntil [condition]
```

SAMPLE CODE AND OUTPUT

```
to mwait :msecs
resett
waituntil [timer > :msecs]
end
```

when-off

DESCRIPTION

Stops a previously started `when` process.

USAGE FORMAT

```
when-off
```

SAMPLE CODE AND OUTPUT

```
when-off
```

when

DESCRIPTION

The `when` primitive starts a background process which checks a condition between each line of logo code running in the main process. If the condition becomes true, a specific block of code is executed.

USAGE FORMAT

```
when [condition] [code-block]
```

The *condition* is what is checked in the background and *code-block* is what is executed every time the *condition* becomes true.

Note: the when condition is edge-triggered. That is, if the condition becomes true and remains true after the code is executed, the code will not be executed again until the condition becomes false and then true again.

SAMPLE CODE AND OUTPUT

```
when [btst 3 porta]
     [print-string "|Switch pressed!|]
```

SYSTEM COMMANDS

constant

DESCRIPTION

Declares constant variables that are replaced with their values by the compiler at compile time.

USAGE FORMAT

```
constant [constant-list]
```

Note: The declarative `constant` should be used along with procedure definitions in the source code and cannot be used in the command-center.

SAMPLE CODE AND OUTPUT

```
Constant [[a 2][b 3]]
```

write-reg

DESCRIPTION

Short for **"write register"**, writes to a PIC register.

USAGE FORMAT

write-reg ***address*** ***value***

The argument *address* is the address of a PIC register to which we want to write and *value* is the value to write.

SAMPLE CODE AND OUTPUT

```
to ad :chan
write-reg adcon1 $80
write-reg adcon0 ((:chan - 1) * 8) + $81
bset adgo adcon0
waituntil [not btst adgo adcon0]
output ((read-reg adresh) * 256) + read-reg adresl
end
```

read-reg

DESCRIPTION

Short for **”read-reg”**, examines and returns the content of one of the internal PIC registers.

USAGE FORMAT

`read-reg` ***`address`***

The argument *`address`* is the address of the PIC register we want to examine.

SAMPLE CODE AND OUTPUT

```
print read-reg 5
>(the conent of register with address 5)
```

read-prog-mem

DESCRIPTION

Reads and returns a byte from program flash eeprom.

USAGE FORMAT

read-prog-mem ***address***

The argument ***address*** is the program memory location from which we want to read.

SAMPLE CODE AND OUTPUT

```
to print-string :n
setnn :n
loop
[if (read-prog-mem nn) = 0 [stop]
put-serial read-prog-mem nn
setnn nn + 1]
end
```

global

DESCRIPTION

Returns the value of a global.

USAGE FORMAT

```
global    pointer
```

The argument ***pointer*** is a pointer to the global.

Note 1: In most cases, it is easier to get a global variable's value by just using its name.

Note 2: The primitive *global* is only useful when using pointers to globals. This is done using macros defined when you define a global variable. For example, if you define a global using global [foo] then the macro *foo is defined as a pointer to the global variables. Globals are stored sequentially as they are declared, two bytes apiece. Thus the pointers are also sequential, increasing by two each time.

SAMPLE CODE AND OUTPUT

```
global [foo]

setfoo 15
print foo
>15
```

on-startup

DESCRIPTION

Declares what needs to run when the Tower is first turned on or power-cycled.

USAGE FORMAT

```
on-startup [command-list]
```

Note: The declarative `on-startup` should only be used along with procedure definitions in the source code and cannot be used in the command-center.

SAMPLE CODE AND OUTPUT

```
on-startup [your-favorite-startup-procedure-and/or-
                              list-of-commands]
```

on-white-button

DESCRIPTION

Declares what commands should run when the white button is pressed.

USAGE FORMAT

```
on-white-button [command-list]
```

Note 1: The declarative `on-white-button` should only be used along with procedure definitions in the source code and cannot be used in the command-center.

Note 2: If the Tower is already running a program (indicated by a pulsating blue LED), pressing the white button stops the program. When the white button is pressed for a second time, the Tower runs the list of commands declared by the `on-white-button` declarative.

SAMPLE CODE AND OUTPUT

```
on-white-button [your-favorite-run-procedure-and/or-
                          list-of-commands]
```

resett

DESCRIPTION

Resets the timer to zero.

USAGE FORMAT

```
resett
```

Note: Affects only the value that the `timer` primitive reports.

SAMPLE CODE AND OUTPUT

```
resett wait 10 print timer
>100
```

setglobal

DESCRIPTION

Sets the value of a global.

USAGE FORMAT

```
setglobal pointer new-value
```

The argument *pointer* is the pointer to the global and the argument *new-value* is the new value to be assigned to the global variable.

Note 1: In most cases, this is easier to use *set* and the name of the global to set the value of the global; for example, using *global [foo]* to declare a global, you can use *setfoo* to set its value. Note that globals are declared along with the procedure definitions in the source code and cannot be used in the command-center.

Note 2: The primitive *setglobal* itself is only useful when you use pointers to globals. This is done using macros defined when you define a global variable. For example, if you define a global using *global [foo]* then the macro **foo* is defined as a pointer to the global variables. Globals are stored sequentially as they are declared, two bytes apiece. Thus the pointers are also sequential, increasing by two each time.

SAMPLE CODE AND OUTPUT

```
global [foo]

setglobal *foo 14
print foo
>14
setglobal *foo + 2 1234
print global *foo + 2
>1234
```

timer

DESCRIPTION

Returns the current value of the timer.

USAGE FORMAT

```
timer
```

Note: The timer overflows after 32767 milliseconds and resets to zero. (It never has a negative value.)

SAMPLE CODE AND OUTPUT

```
resett wait 10 print timer
>100
```

PIN CONTROL

flip-bit

DESCRIPTION

Toggles a bit on an internal PIC register.

USAGE FORMAT

flip-bit *bit-number register*

The argument *bit-number* is the bit number (0-7) of the PIC register, with address *register,* that you want toggle.

Note: If the register corresponds to a PIC i/o port, the bit number would correspond to the pin number on the port.

SAMPLE CODE AND OUTPUT

```
to toggle :chan :port
clear-bit :chan (:port + $80)
flip-bit :chan :port
end
```

clear-bit

DESCRIPTION

Clears a bit on an internal PIC register.

USAGE FORMAT

clear-bit *bit-number register*

The argument *bit-number* is the bit number (0-7) of the PIC register, with address *register*, that you want clear.

Note: If the register corresponds to a PIC i/o port, the bit number would correspond to the pin number on the port.

SAMPLE CODE AND OUTPUT

```
to clear :chan :port
clear-bit :chan (:port + $80)
clear-bit :chan :port
end
```

set-bit

DESCRIPTION

Sets a bit on an internal PIC register.

USAGE FORMAT

set-bit *bit-number register*

The argument *bit-number* is the bit number (0-7) of the PIC register, with address *register*, that you want set.

Note: If the register corresponds to a PIC i/o port, the bit number would correspond to the pin number on the port.

SAMPLE CODE AND OUTPUT

```
to set-bit :chan :port
bclr :chan (:port + $80)
bset :chan :port
end
```

test-bit

DESCRIPTION

Returns true if the bit on an internal PIC register is set and false otherwise.

USAGE FORMAT

test-bit *bit-number register*

The argument *bit-number* is the bit number (0-7) of the PIC register, with address *register*, that you want test.

Note: If the register corresponds to a PIC i/o port, the bit number would correspond to the pin number on the port.

SAMPLE CODE AND OUTPUT

```
print test-bit 1 5
>(prints a 1 if bit 1 of register 5 is set)
>(prints a 0 if bit 1 of register 5 is clear)
```

COMMUNICATION

get-serial

DESCRIPTION

Returns the last byte received on the serial port, or –1 if no byte has been received.

USAGE FORMAT

```
get-serial
```

SAMPLE CODE AND OUTPUT

```
waituntil [new-serial?] print get-serial
```

i2c-read-byte

DESCRIPTION

Reads a specified number of bytes from the i2c bus.

USAGE FORMAT

`i2c-read-byte last-byte-argument`

Use a 0 for *`last-byte-argument`* when asking an i2c slave for the last byte (and when asking for just one byte) and a 1 otherwise.

SAMPLE CODE AND OUTPUT

```
to sensor :n
i2c-start
i2c-write-byte $0a
i2c-write-byte 2
i2c-write-byte 0
i2c-write-byte (:n - 1)
i2c-stop
i2c-start
i2c-write-byte $0b
ignore i2c-read-byte 1
seti2c-byte (lsh i2c-read-byte 1 8)
seti2c-byte i2c-byte or i2c-read-byte 0
i2c-stop
output i2c-byte
end
```

i2c-start

DESCRIPTION

Starts an i2c communication sequence.

USAGE FORMAT

```
i2c-start
```

SAMPLE CODE AND OUTPUT

```
to sensor :n
i2c-start
i2c-write-byte $0a
i2c-write-byte 2
i2c-write-byte 0
i2c-write-byte (:n - 1)
i2c-stop
i2c-start
i2c-write-byte $0b
ignore i2c-read-byte 1
seti2c-byte (lsh i2c-read-byte 1 8)
seti2c-byte i2c-byte or i2c-read-byte 0
i2c-stop
output i2c-byte
end
```

i2c-stop

DESCRIPTION

Ends an i2c communication sequence.

USAGE FORMAT

```
i2c-stop
```

SAMPLE CODE AND OUTPUT

```
to sensor :n
i2c-start
i2c-write-byte $0a
i2c-write-byte 2
i2c-write-byte 0
i2c-write-byte (:n - 1)
i2c-stop
i2c-start
i2c-write-byte $0b
ignore i2c-read-byte 1
seti2c-byte (lsh i2c-read-byte 1 8)
seti2c-byte i2c-byte or i2c-read-byte 0
i2c-stop
output i2c-byte
end
```

i2c-write-byte

DESCRIPTION

Writes a byte on the i2c bus.

USAGE FORMAT

i2c-write-byte *byte*

The argument *byte* is the value to be sent on the i2c bus.

Note: Look at the example below to learn what *byte* represents depending on the argument to which *i2c-write-byte* command after the i2c-start it is. The first *i2c-write-byte* command takes the address of the i2c-slave as an argument. The argument to each subsequent *i2c-write-byte* commands, prior to an *i2c-stop* command, depend on the functionality of the i2c slave board.

SAMPLE CODE AND OUTPUT

```
to sensor :n
i2c-start
i2c-write-byte $0a
i2c-write-byte 2
i2c-write-byte 0
i2c-write-byte (:n - 1)
i2c-stop
i2c-start
i2c-write-byte $0b
ignore i2c-read-byte 1
seti2c-byte (lsh i2c-read-byte 1 8)
seti2c-byte i2c-byte or i2c-read-byte 0
i2c-stop
output i2c-byte
end
```

new-serial?

DESCRIPTION

Returns true if a byte has been received on the serial port since the last time the *new-serial?* command was issued.

USAGE FORMAT

```
new-serial?
```

SAMPLE CODE AND OUTPUT

```
waituntil [new-serial?] print get-serial
```

put-serial

DESCRIPTION

Sends a byte over the serial port.

USAGE FORMAT

put-serial *value*

The argument *value* is the byte to send over serial.

Note: When the byte is sent back to the TDE, it will print the ASCII character corresponding to the byte sent.

SAMPLE CODE AND OUTPUT

```
print put-serial 65
>A
```

setbaud-2400

DESCRIPTION

Sets the serial communication rate to 2400 bps.

USAGE FORMAT

setbaud-2400

Note: This the default baud rate for the serial communication.

SAMPLE CODE AND OUTPUT

```
Setbaud-2400
>
```

setbaud-9600

DESCRIPTION

Sets the serial communication rate to 9600 bps.

USAGE FORMAT

setbaud-9600

Note: This the default baud rate for the serial communication is 2400.

SAMPLE CODE AND OUTPUT

```
Setbaud-9600
>
```

ARITHMETIC AND LOGIC

arithmetic

+, -, *, /, %

DESCRIPTION

Returns the result of arithmetical operators (*+*, *-*, *, */*) applied to 16-bit numerical operands.

USAGE FORMAT

num1 (+, -, *, /, %) num2

Note: The */* operator returns the integer part of the result if dividing *num1* by *num2*. The *%* operator returns the remainder of a dividing *num1* by *num2*.

SAMPLE CODE AND OUTPUT

```
print 4 + -10
>-6

print 42 - 8
>34

print 4 * -10
>-40

print 35 / 10
>3

print 35 % 10
>5
```

comparisons

=, >, <

DESCRIPTION

Returns a Boolean depending on the result of arithmetical comparison operators (=, <, >) applied to 16-bit numerical operands.

USAGE FORMAT

```
num1 (=, >, <) num2
```

SAMPLE CODE AND OUTPUT

```
print 10 = 10
>1

print 4 > 2
>1

print 4 < 2
>0
```

bit-wise logic

and, or, xor

DESCRIPTION

Returns the result of bit-wise logical operators (*and*, *or*, *xor*) applied to 16-bit numerical operands.

USAGE FORMAT

```
num1 (and, or, xor) num2
```

SAMPLE CODE AND OUTPUT

```
print 20 and 4
>4

print 16 or 4
>20

print 7 xor 3
>4

print (1 = 1) and (1 = 0)
>0

print (1 = 1) or (1 = 0)
>1

print (1 = 1) or (1 = 0)
>1
```

highbyte

DESCRIPTION

Returns the high byte of a 16 bit value.

USAGE FORMAT

```
highbyte num
```

SAMPLE CODE AND OUTPUT

```
print highbyte $4000
>64
```

lowbyte

DESCRIPTION

Returns the low byte of a 16 bit value.

USAGE FORMAT

```
lowbyte num
```

SAMPLE CODE AND OUTPUT

```
print lowbyte $6560
>96
```

lsh

DESCRIPTION

Used to left shift or right shift a number by a specified number of bits.

USAGE FORMAT

lsh ***num dist***

The argument *num* is the number to shift and *dist* is the numbers bits by which to shift *num*.

Note: A negative value for *dist* represents a right shift, while a positive value for *dist* represents a right shift.

SAMPLE CODE AND OUTPUT

```
print lsh 4 1
>8

print lsh 64 -2
>16
```

random

DESCRIPTION

Returns a random number between 0 and 32767, inclusive.

USAGE FORMAT

```
random
```

SAMPLE CODE AND OUTPUT

```
print random
>5622
```

not

DESCRIPTION

Returns a Boolean ***not*** of a number.

USAGE FORMAT

```
not num
```

Note: This operation is a Boolean ***not*** and not a bit-wise ***not***.

SAMPLE CODE AND OUTPUT

```
print not 42
>0

print not 0
>1
```