LogoChip Logo Language Reference (1/03)

1. Overview

LogoChip Logo is the a programming environment for the LogoChip, a minimal set of hardware is based on a PIC16F876 microcontroller (See Appendix).

LogoChip Logo has the following features:

- 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 mulitasking `when` primitive
- a 16-bit number system (addition, subtraction, multiplication, division, comparison);
- timing functions and a random number function

When using LogoChip Logo, user programs are entered on a desktop computer and compiled into tokens which are transferred to the LogoChip through the serial port of the desktop. Logo commands can be executed by typing a line in the LogoChip Logo -**command center-** and pressing the <ENTER> key.

LogoChip 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 the LogoChip either by

1) Placing them in a text file located in the same folder as the LogoChip software, typing the textfile name in the textbox in the lower left corner of the -**command center-** and clicking on the **compile-file** button

or

2) (pc version only) clicking on the –**procs-** button in the lower right hand corner of the command center to get to the **-procedures-** page, and clicking on the **download** button. (Click on the **-cc-** button to return to the command center.)

When the LogoChip is idle, pressing its **start-stop** push-button causes it to begin executing the most recent command that has been executed from the LogoChip Logo -**command center--**. When the LogoChip is running a program, the indicator LED on the LogoChip changes from red to green. Pressing the **start-stop** button on the LogoChip causes it to halt program execution.

2. Setting and Reading Registers

The PIC16F876 has the register structure shown in the appendix of this document.. These registers can be set and read using the following primitives.

`write` *`register-address value`* writes a one byte *value* in the register whose address is *register-address*.

Example:

```
constants [portb 6]
write portb 68          ;writes the number 68 into the portb
                        register
```

`read` *`register-address`* reports the one byte *value* contained in the register whose address is *register-address*.

`setbit` *`bit-number register-address`* sets (makes HIGH) the *bit-number*th bit of the register whose address is *register-address*.

Example:

```
constants [portb 6]
setbit 5 portb          ;sets bit 5 of the portb register
                        HIGH
```

`clearbit` *`bit-number register-address`* clears (makes LOW) the *bit-number*th bit of the register whose address is *register-address*.

`togglebit` *`bit-number register-address`* toggles the *bit-number*th bit of the register whose address is *register-address*.

Example:

```
constants [portb 6]
togglebit 0 portb       ;if bit 0 of the portb register is
                        set then this command will clear it,
                        if it is cleared then this command
                        will set it.
```

`testbit` *`bit-number register-address`* reports true if the *bit-number*th bit of *`register-address`* is set, reports false if it is cleared.

Example:

```
constants [portc 7]
waituntil [testbit 0 portc]   ;waits until bit 0 of portc is
                              set.
```

`leftshift` *`num1 num2`* reports *num1* shifted by *num2* bits. If *num2* > 0 then the number is shifted to the left, if *num2* < 0 then the number is shifted to the right. Useful for pulse width modulation.

Example:

```
setn 11
loop [
     ifelse ($80 and n)  ;test the value of the MSB of n
          [setbit 0 portb]
          [clearbit 0 portb]
setn leftshift n 1
     ]
```

`read-rom` *`rom-address`*

("examine read only memory,,) reports the one byte value contained in the ROM location whose address is *`rom-address.`* Used for examining the logo program byte codes (which are stored starting at rom-address $480).

3. Timing

The timing commands are useful to cause the LogoChip to do something for a length of time.

Example:

```
constants [portb 6]
setbit 1 portb          ;sets bit 1 of portb for two seconds
                        and then clears it.
wait 20
clearbit 1 portb
```

Please note that there are two different reference values for timing: 0.1 second units, used in `wait` , and 0.001 second units, used in `timer`.

`wait` *`duration`*

Delays for a duration of time, where *duration* is given in tenths-of-seconds. *E.g.*, `wait 10` inserts a delay of one second.

`timer`

Reports value of free-running elapsed time device. Time units are reported in 1 millisecond counts.

`resett`

Resets elapsed time counter to zero.

`no-op`

("no-operation,,) does nothing, takes about 70 microseconds to execute each `no-op` command. Useful for inserting short delays.

Example:

```
to short-wait :n                ; a delay of about 70 * :n
                                microseconds
     repeat :n [no-op]
end
```

4. Input/Output

`flash`

causes the red/green indicator LED to flash 5 times

The LogoChip has 17 pins available to the user for input and output. These are PIC pins A0-A5, B0-RB7, and C0-C2.

Each time the LogoChip is turned on , pins A4, B0-B7, and C0-C2 are initially configured as digital inputs while pins A0-A3 and A5 are configured as analog inputs. All of these pins can be configured changed to digital outputs through use of the corresponding bit in the PIC's data direction registers. (Note however that when configured as an output pin A4 is an "open collector,, output so it needs a pullup resistor to function properly.)

Example:

```
constants [[portb 6] [portb-ddr $86]]
to set-pinB2-high
clearbit 2 portb-ddr     ;clears bit 2 of the portb data
                         direction register, which turns pin
                         B2 into an output
setbit 2 portb           ;sets bit 2 of portb HIGH, making the
                         B2 pin go to +5V.
```

Pins A0-A3 and A5 are initially configured as 10-bit analog to digital converters.

`read-ad` *num* — reports the 10 bit digital value corresponding to the voltage level on a channel *num* of porta.

5. Control

Chip Logo supports the following control structures:

`loop [`*body*`]` — Repetitively executes *body* indefinitely

`repeat` *times* `[`*body*`]` — Executes *body* for times repetitions. *times* may be a constant or calculated value.

`if` *condition* `[`*body*`]` — If *condition* is true, executes *body*. Note: a condition expression that evaluates to zero is considered "false,,; all non-zero expressions are "true,,.

`ifelse` *condition* `[`*body-1*`]` *[body-2]*

If *condition* is true, executes *body-1*; otherwise, executes *body-2*.

`waituntil [`*condition*`]`

	Loops repeatedly testing *condition*, continuing subsequent program execution after it becomes true. Note that *condition* must be contained in square brackets; this is unlike the conditions for `if` and `ifelse`, which do not use brackets.
`stop`	Terminates execution of procedure, returning control to calling procedure.
`output` *value*	Terminates execution of procedure, reporting *value* as result.

6. Multitasking

Chip Logo contains a single `when` primitive that allows for simple multitasking:

`when [`*condition*`] [`*body*`]`

	Launches a parallel process that repeatedly checks *condition* and executes *body* whenever *condition* changes from false to true. The `when` rule is "edge-triggered,, and remains in effect until it is turned off with the `whenoff` primitive. Only one `when` rule can be in effect at a time; if a new `when` rule is executed by the program, this new rule replaces the previous rule.
`whenoff`	Turns off any existing `when` rule.

For example, the following program will toggle bit 0 of portb every second , while constantly monitoring bit 1 of portc.

```
to walk-and-chew-gum
resett
when [timer > 1000] [bchg 0 portb resett]
loop [
     ifelse (testbit 0 portc)
          [setbit 1 portb]
          [clearbit 1 portb]]
end
```

Only one `when` rule can be active at a time.

7. Numbers

LogoChipLogo uses 16-bit integers between -32768 and + 32767.

All arithmetic operators must be separated by a space on either side. E.g., the expression `3+4` is not valid. Use `3 + 4`.

`+`	Infix addition.
`-`	Infix subtraction.
`*`	Infix multiplication
`/`	Infix division.
`%`	Infix modulus (remainder after integer division).
`and`	Infix logical "and,, operation (bitwise and).
`or`	Infix logical or operation (bitwise or).
`not`	Prefix logical not operation. Unlike the `and` and `or` primitives, `not` is not a bitwise operation. If *num* is any non-zero integer (corresponding to a logical true) then `not` *num* evaluates as zero (logical false.). If *num* is zero then `not` *num* evaluates as one (logical true).
`random`	Reports a pseudo-random number from 0 to 32767.
`lowbyte` *number*	Reports the low order byte of a 16-bit *number*.
`highbyte` *number*	Reports the high order byte of a 16-bit *number*.

Putting a "$,, in front of a number (without a space!) causes LogoChip Logo to treat the number as a hexadecimal value.

8. Global Variables and Constants

There are two built-in global variables called `m` and `n`. The commands `setm` and `setn` are used to set the value of these variables. For example

```
setn 5
```

will set the value of `n` to 5.

Additional global variables are created by including the `global [`*variable-list*`]` directive along with the procedures definitions. *E.g.*,

```
global [foo bar]
```

creates two additional globals, named `foo` and `bar`. Additionally, two global-setting primitives are created: `setfoo` and `setbar`. Thus, after the global directive is interpreted, one can say

```
setfoo 3
```

to set the value of foo to 3, and

```
setfoo foo + 1
```

to increment the value of `foo`.

There is a limit of a maximum of 15 different global variables (including `n` and `m`) that can be used in a LogoChip Logo program.

Constants can be declared by including the `constants [`*`constant-list`*`]` directive along with the procedures definitions. *E.g.*,

```
constants [[portb 6] [portb-ddr $86]]
```

will cause the LogoChip Logo compiler to substitute the number 6 in place of each use of the word `portb` the number $86 in place of each use of the word `portb-ddr` that appears in a user program.

9. Procedure Inputs and Outputs

Procedures can accept arguments using Logo's colon syntax. *E.g.*,

```
to flash :times
repeat :times [setbit 0 6 wait 2 clearbit 0 6]
end
```

creates a procedure named `flash` that takes an input which is used as the counter in a repeat loop.

Procedures may return values using the `output` primitive; *e.g.*:

```
to go
repeat ntimes [setbit 0 6 wait 2 clearbit 0 6]
end
to ntimes
ifelse testbit 0 7 [output 1][output 3]
end
```

The `go` procedure will execute 1 or 3 times depending on the value of bit 0 of portc.

10. Recursion

Chip Logo supports tail recursion to create infinite loops. For example:

```
to beep-forever
beep wait 1
beep-forever
```

```
end
```

is equivalent to

```
to beep-forever
loop [beep wait 1]
end
```

The recursive call must appear as the last line of the procedure and cannot be part of a control structure like `if` or `waituntil`. Thus the following is **not** valid:

```
to beep-when-pressed
beep wait 1
if switcha [beep-when-pressed]
end
```

11. Communication

send *value* — transmits an 8-bit *value* to the desktop computer via the serial connection

Example:

This `print` procedure defined below makes use of send to prints the 16-bit value of the monitor box on the LogoChip Logo screen. (A copy of `print` procedure is currently included in the `lctools` file that can be found in the LogoChip Logo software folder.)

```
to print :n
if :n < 0 [send 45 pn 0 - :n stop]
pn :n
cr
end

to pn :n
if :n > 9999 [pdigit :n 10000]
if :n > 999 [pdigit :n 1000]
if :n > 99 [pdigit :n 100]
if :n > 9 [pdigit :n 10]
pdigit :n 1
end

to pdigit :n :d
send ((:n / :d) % 10) + 48
end

to cr
send 13
```

```
end
```

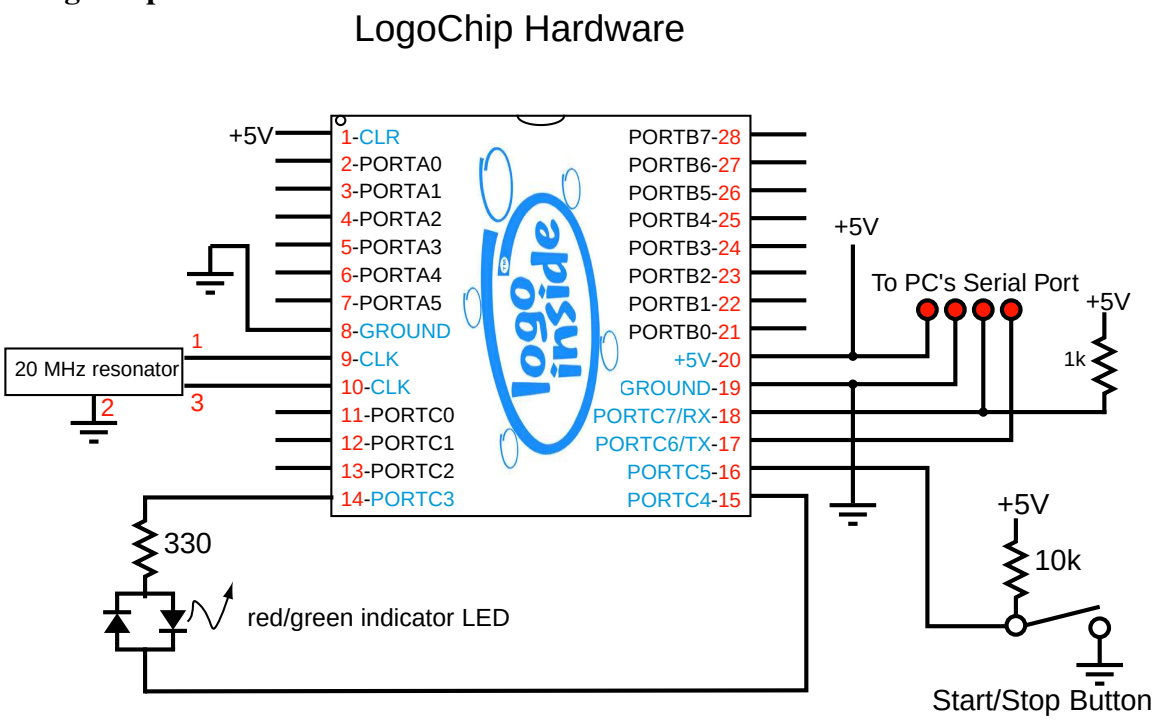
12. Limitations

The maximum size of a Chip Logo program running on aPIC16F876 is 2944? bytes. When a program is downloaded, its size is displayed in the “status box„ near the bottom of the Logo Chip Logo procedures window.

The LogoChip has a limited stack size the number of nested procedure calls should be kept below about 7. When there is a stack overflow the LogoChip’s red/green LED will flash red and green and the program will stop running. (The flashing lasts three times as long as an ordinary “flash„.)

Appendix:

LogoChip Hardware



PIC Register Map

The diagram on the next page shows a map of the PIC's file registers. Most of these are used by the LogoChip virtual machine and should not be directly altered by LogoChip Logo code. The most important registers for use in LogoChip Logo code are the PORT registers

PORTA ($05), PORTB ($06), and PORTC ($07).

and the corresponding data direction registers

TRISA ($85) TRISB ($86) and TRISC ($87).

Clearing a bit in a data direction register turns the corresponding bit into an output, setting a bit in a data direction register turns the corresponding bit into an input.

The other registers that are available for use by LogoChip Logo programs are[1]:

ADRESH ($1e), ADCON0 (1f), ADRESL ($9e), ADCON1 ($9f) ;for use with the PIC's built-in analog to digital conversion module.

T2CON ($12), CCP1CON ($17), CCPR1L ($15), PR2 ($92) ;for use with the PIC's built-in pulse width modulation module.

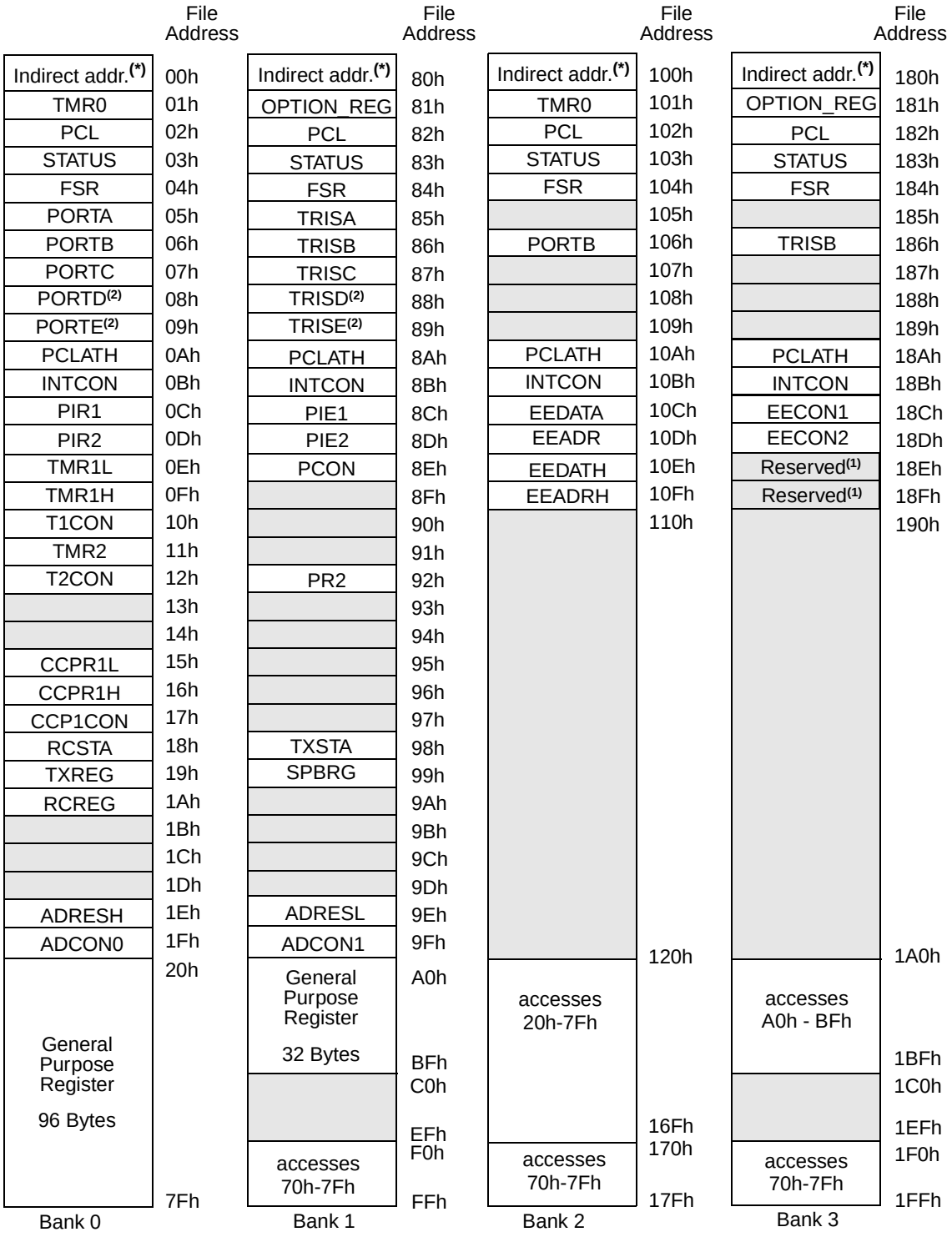
TMR1L ($0E), TMR1H ($0F), TMR1CON ($10) ;for use with TIMER1, the PIC's built-in 16-bit counter/timer module.

TMR2 ($11), T2CON ($12) ;for use with TIMER2, the PIC's built-in 8-bit counter/timer module.

Most other registers are used by the LogoChip Virtual Machine and should not be written to!!

[1] See the PIC16F870 datasheet for more details regarding use of these special registers.

PIC16F870/871 REGISTER FILE MAP



Unimplemented data memory locations, read as '0'.

* Not a physical register.

Note 1: These registers are reserved; maintain these registers clear.

2: These registers are not implemented on the PIC16F870.

PIC Program Memory Map

The LogoChip has 2k of 14-bit EEPROM (electrically erasable programmable read only memory) for storing the PIC's program.

(The machine code generally consists of a 6 bit OPCODE part, which specifies which type of instruction it is, and an OPERAND, which provides specific information to be used in the instruction. For example the following command will store the contents of the accumulator in the specified file register.

```
          [sta x]
     store accumulator - The contents of the accumulator are loaded
into register "f", where "f" has a value from 0 through 127. (The
value of the accumulator remains unchanged.)

          status flags affected:  Z

Coding for storing accumulator in regester:
```

00	0000	1fff	ffff

.)

The addresses of these memory locations are $00-$7ff. In the LogoChip this memory is allocated as follows:

$00-$7f – is for the "monitor,, which,when the PIC is started in "bootload mode,,(by holding the button down), is responsible for programming new PIC programs into the memory space between $80 and $47f. When not started in bootload mode the execution simply jumps to location $80.

$80 - $47F – is for the LogoChip Virtual Machine. Assembly language programs can also be written into this space, overwriting the virtual machine allowing the to run ordinary PIC code, while still maintaining the monitor.

$480 - $7ff – This is where user Logo byte codes are stored. As the name implies, the byte codes are only 8 bits wide. The upper 6 bits of each memory location are note used.