LogoChip Logo Language Reference v. 2.0

Overview

LogoChip Logo is the a programming environment for the LogoChip, a minimal set of hardware is based on a Microchip PIC18F2320 microcontroller[1].

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 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 a serial connection. Logo commands can be executed by typing a line in the LogoChip Logo command center and pressing the <ENTER> key. The maximum size of a LogoChip Logo program running on a PIC18F2320 microcontroller is 5k bytes. When a program is downloaded, its size is displayed in the “status box” near the bottom of the Logo Chip Logo procedures window.

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 by placing them in a text file , typing the text file name in the textbox in the lower left corner of the command center and clicking on the **download** button.

[1] LogoChip Logo can also PIC18F4320 microcontroller, which has more input / output pins than a PIC18F2320.

The LogoChip has the ability to run programs in the absence of a desktop computer. A pair of special procedures called “startup” and “powerup” can be defined by the user The powerup procedure will execute when the LogoChip is turned on via its on/off switch. The startup procedure will execute if the LogoChip is in its idle state and the start/stop button is pressed.

Reading and Writing Registers

The PIC18F2320 microcontroller has a collection of 640 byte-wide registers with addresses in the range 0-511 ($000-$1ff) and also 3968-4095 ($f80-$fff). These registers are implemented as static RAM and include both general purpose registers, which LogoChip Logo uses for its stack operations and to store global variables, and also special function registers, which are used by the microcontroller’s central processing unit and peripheral modules for controlling the desired operation of the device. All of these registers can be set and read using the LogoChip Logo primitives described below. For example:

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

Example:

```
    write $f81 68            ; writes the number 68 into the
                             register whose address is $f81 (the
                             portb register)
```

The ability to directly write and read all microcontroller registers is central to design LogoChip design philosophy, giving the user access to much of the microcontroller’s functionality. However care must be exercised, since many of these registers are used by the Logo virtual machine and altering any of these registers is likely to cause the Logo virtual machine to crash. Only a subset of these many registers should be written to by a LogoChip Logo program. A listing of some of the most commonly used registers and their addresses is shown in the table below.[2]

[2] A complete register map for the PIC18F2320 microcontroller, along with an explicit listing of which registers can be written to a LogoChip Logo program without interfering with the operation of the virtual machine, is given in the *Register Map* section of the appendix.

register name	register address	register function
porta	$f80	porta data register
porta-ddr	$f92	porta data-direction register
portb	$f81	portb data register
portb-ddr	$f93	portb data-direction register
portc	$f82	portc data register
portc-ddr	$f94	portc data-direction register
portd *	$f83	portd data register
portd-ddr *	$f95	portd data-direction register
porte *	$f84	porte data register
porte-ddr *	$f96	porte data-direction register

* Only available in the 40 pin version of the LogoChip based on the PIC18F4320 microcontroller

For each of the commonly used registers listed in the above table LogoChip Logo the register name has been implemented as a predefined constant that returns the address of the register. So alternatively one can write to portb using:

```
    write portb 68            ; writes the number 68 into the portb
                              register
```

The other primitives for reading and writing to registers are:

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:

```
     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:

```
     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:

```
     waituntil [testbit 3 portb]    ;waits until bit 3 of portb is
                                    set.
```

In addition to the file registers discussed above, the PIC18F2320 has 8k bytes of flash memory, which is a type of EEPROM (electrically erasable programmable read only memory). The contents of the flash memory can be read using:

read-rom *rom-address*

("read read–only–memory") reports the 16-bit value obtained from the consecutive bytes stored in ROM locations `rom-address` and `rom–address + 1`.

The addresses of these memory locations are $0000-$1fff. In the LogoChip this memory is allocated as described in the *Flash Program Memory* section of the appendix.

Timing

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

Example:

```
     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 `mwait` and `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.

`mwait` *`duration`*

Delays for a duration of time, where *duration* is given in milliseconds. *E.g.*, `mwait` 53 inserts a delay of 53 milliseconds.

`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 13 microseconds to execute each `no-op` command. Useful for inserting short delays.

Example:

```
     to short-wait :n                ; a delay of about 13 * :n
                                     microseconds
          repeat :n [no-op]
     end
```

Input/Output

`flash` — causes the red/green indicator LED to flash red and green 5 times.

The red/green indicator LED should also flash when the LogoChip powers up. The indicator light is steady red when the LogoChip is powered up and idle and steady green when the LogoChip is running a program.

The LogoChip has 17 pins available to the user for input and output. These are the pins labeled A0-A5, B0-B7, and C2, C6, and C7.

Each time the LogoChip is turned on, pins A4, B0-B7, and C2, C6, and C7 are initially configured as digital inputs. All of these pins can be configured changed to digital outputs through use of the corresponding bit in the microcontroller’s data direction registers. (Note however that when configured as an output pin A4 is an “open collector” output so it needs a pull–up resistor to function properly.)

Example:

```
     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 +Vcc.
```

Pins A0-A3 and A5 are configured as 10-bit analog to digital converters. Analog values can be read using:

`read-ad` *`num`* — reports the 10 bit digital value corresponding to the voltage level on a channel *num* of porta. (Pins A0-A3 correspond to channels 0 through 3, respectively, while pin A5 corresponds to channel 4.)

Control

LogoChip 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.

stop!

Terminates execution completely.

Recursion

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

```
to flash-forever
flash wait 1
flash-forever
end
```

is equivalent to

```
to flash-forever
loop [flash 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`. Thus the following program, which attempts to cause a persistent flash on the indicator LED as long as pin B0 is high, is **not** valid:

```
to flash-when-b0-is-high
flash
if (testbit 0 portb) [flash-when-b0-is-high]
end
```

Numbers

LogoChip Logo 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).

`xor`

Infix logical “xor” operation (bitwise xor).

`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.

Example: The following procedure will report a pseudo-random number between 0 and 100.

```
to random100
output random % 100
end
```

`lowbyte` *`number`*

Reports the low order byte of a 16-bit *`number`*.

`highbyte` *`number`*

Reports the high order byte of a 16-bit *`number`*.

`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. Thus the result is equal to *num1* * 2^{num2}

Example:

```
leftshift 9 2            reports a 36
leftshift 9 -2           reports a 2
```

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

Putting a "#" in front of a number (without a space!) causes LogoChip Logo to treat the number as a **binary** value.

Procedures

Inputs and outputs

Procedures can accept arguments using Logo's colon syntax. For example,

```
to multi-flash :times
repeat :times [flash wait 10]
end
```

creates a procedure named `multi-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 portb [output 1][output 3]
end
```

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

The startup procedure

If the LogoChip is not running a program (indicator LED is red) and if a procedure called "startup" is contained in the most recently downloaded set of procedures, then pressing the start/stop button will cause the startup procedure to run. This feature enables a LogoChip to run a desired program when it is not connected to a desktop or laptop computer.

If the LogoChip is running a program (indicator LED is green) then pressing the start/stop button will cause the program to stop running.

The powerup procedure

If a procedure called "powerup" is contained in the most recently downloaded set of procedures, then turning on the LogoChip power switch will cause the powerup procedure to run. For example the following procedure will cause all of the pins on portb to be configured as outputs when the LogoChip is turned on.

```
to powerup
write portb-ddr 0
end
```

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" by which we mean that `n` will report a value of 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`.

Please note that the primitives used for reading and writing the contents of registers (`read`, `testbit`, `write`, `setbit` and `clearbit`) are *not* generally used directly with global variables. The register-oriented primitives do not act directly on their arguments, but rather on the registers whose addresses are pointed to by the arguments. For example the program

```
setfoo 255
clearbit 0 foo
```

results in `foo` having a value of 255, not 254. The `clearbit` command causes the zeroth bit of the register whose address is 255 to be zero.

There is a limit of a maximum of 111 different global variables that can be used in a LogoChip Logo program. These variables are numbered with the variable `n` assigned the number 1, the variable `m` assigned the number 2, and subsequent numbers assigned to other named variables in the order in which they are listed in the global declaration. The commands `setglobal` and `global` can be used to write and read the values of any of the global variables:

setglobal *variable-number value*

sets the value of a global variable number *variable-number* to *value*

global *variable-number*

reports the value of a global variable number *variable-number*

These features can be used to implement indexed arrays. For example the following procedures allow the user to set and read values in an array whose elements are stored in global variables numbers 10 and above.

```
to setarray :index :value
setglobal :index + 10 :value
end

to array :index
output global :index + 10
end

setarray 17 4087         ; set the value of array element #17
                         to 4087
print array 17           ; print the value of array element
                         #17 in the monitor box
```

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

```
constants [[num 6] [times 10]]
```

will cause the LogoChip Logo compiler to substitute the number 6 in place of each use of the word `num` and the number 10 in place of each use of the word `times` appears in a user program.

Communication

send *value*

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

Upon powering up, the LogoChip is in a mode where any serial communication sent from the host computer will terminate the current program. In this mode, the LogoChip will respond to any new command is run from the command center.[3]

This `print` procedure defined below makes use of send to prints the 16-bit value of the monitor box on the LogoChip Logo screen.

print *value*

prints a 16-bit value in the monitor box on the right side of the LogoChip Logo screen and moves cursor to the next line

prs "*character-string*

prints the character string that follows the quotation mark in the monitor box on the right side of the LogoChip Logo screen and moves cursor to the next line.

The vertical line character can be used to print strings containing spaces:

```
prs "|hello world|
```

The LogoChip monitor box responds to standard ASCII codes. Thus, for example, the following procedure will generate a carriage return.

```
to cr
send 13
end
```

The LogoChip Tools File

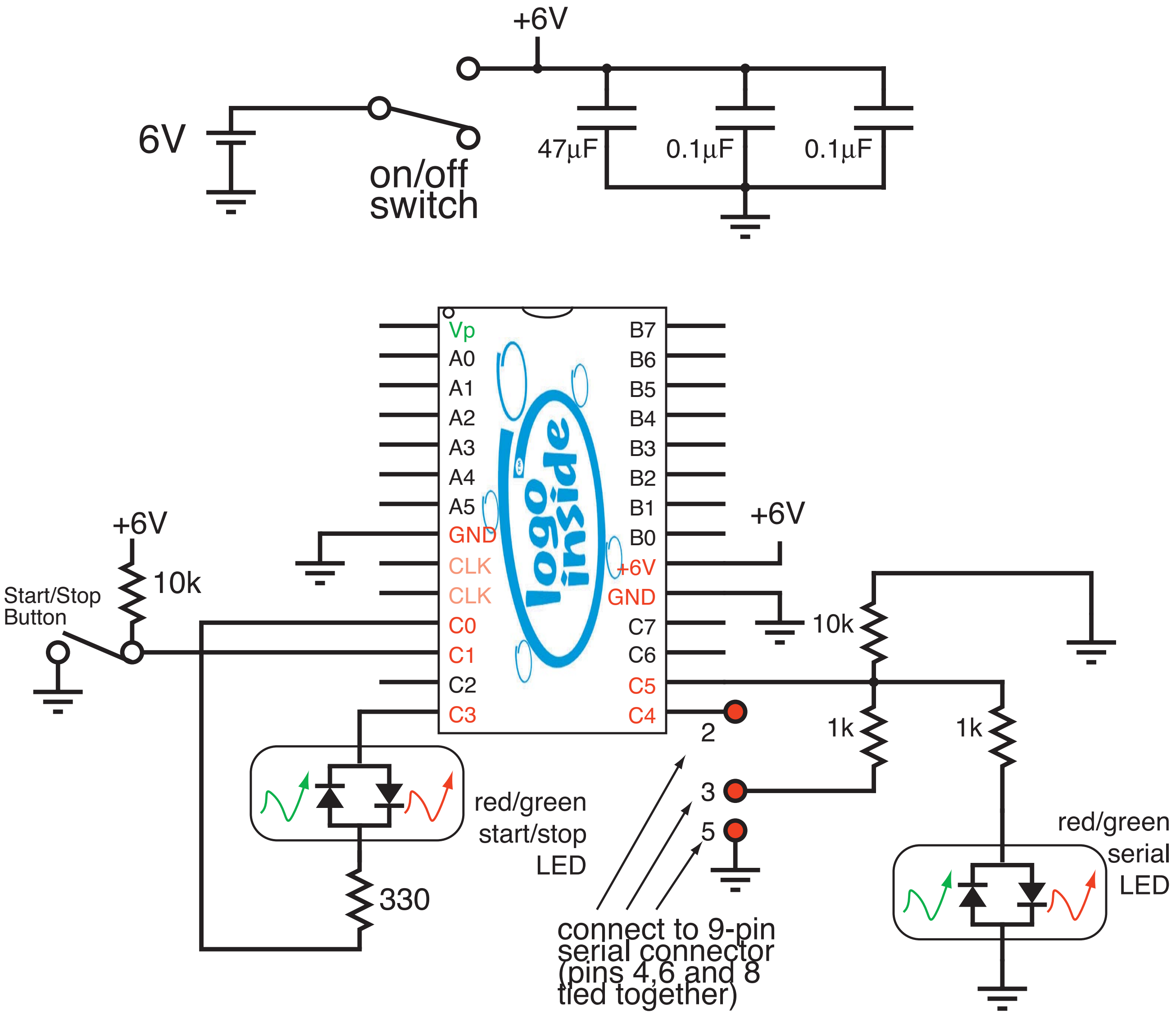
Whenever a text file containing a Logo program is downloaded to the LogoChip, if there is a file called **lc-tools.txt** located in the same folder then the compiler will

[3] The PIC18F2320 microcontroller has a built-in UART that can be used to allow the LogoChip to receive serial communications. See the application note entitled *Using the LogoChip's built-in UART.*

interpret the contents of lc-tools.txt as additional Logo code to be included in the download. This provides a way of providing all programs with access to commonly used procedures.

Appendix

LogoChip Hardware



Register Map

Special Purpose Registers

The diagram below shows a map of the PIC18F2320 microcontroller’s byte-wide special purpose file registers. which have addresses from $f80 through $fff

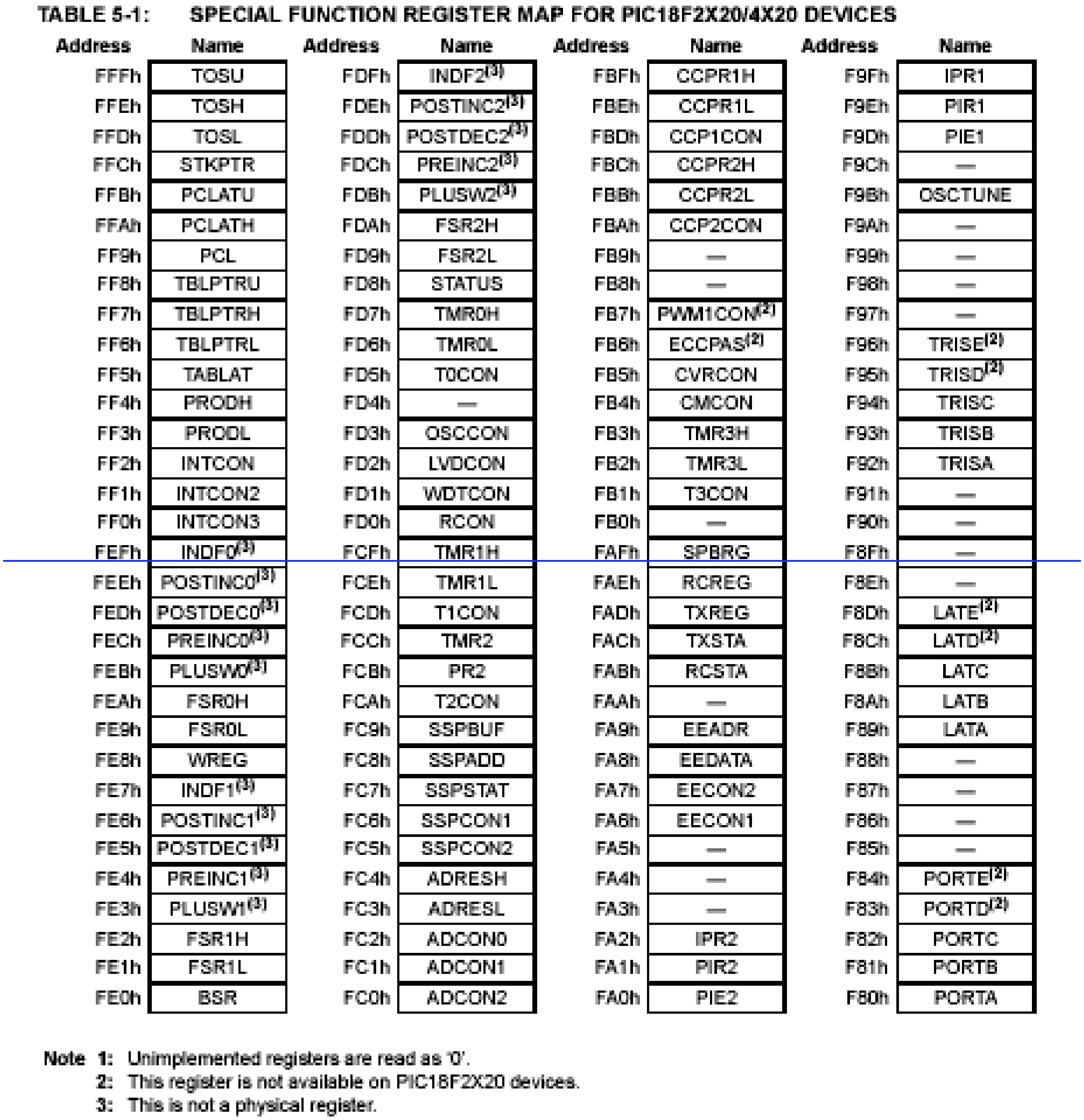


TABLE 5-1: SPECIAL FUNCTION REGISTER MAP FOR PIC18F2X20/4X20 DEVICES

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2[3]	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2[3]	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2[3]	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2[3]	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2[3]	FBBh	CCPR2L	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	PWM1CON[2]	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCPAS[2]	F96h	TRISE[2]
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD[2]
FF4h	PRODH	FD4h	—	FB4h	CMCON	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0[3]	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC0[3]	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0[3]	FCDh	T1CON	FADh	TXREG	F8Dh	LATE[2]
FECh	PREINC0[3]	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD[2]
FEBh	PLUSW0[3]	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1[3]	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1[3]	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1[3]	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1[3]	FC4h	ADRESH	FA4h	—	F84h	PORTE[2]
FE3h	PLUSW1[3]	FC3h	ADRESL	FA3h	—	F83h	PORTD[2]
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

Note 1: Unimplemented registers are read as '0'.
2: This register is not available on PIC18F2X20 devices.
3: This is not a physical register.

The following special registers are used by the Logo Virtual Machine and should *not* be written to:

Register Name	**Register Address**
pir1	$f9e
txsta	$fac
txreg	$fad
rcreg	$fae
adcon1	$fc1
adcon0	$fc2
adresl	$fc3
adresh	$fc4
t2con	$fca
pr2	$fcb
tmr2	$fcc
status	$fd8
spl	$fd9
sph	$fda
@sp+a	$fdb
+@sp	$fdc
@sp-	$fdd
@sp	$fdf
a0l	$fe1
a0h	$fe2
@a0+a	$fe3
+@a0	$fe4
@a0-	$fe5
@a0+	$fe6
@a0	$fe7
acc	$fe8
prodl	$ff3
prodh	$ff4
tablat	$ff5
ipl	$ff6
iph	$ff7
pcl	$ff9
pclath	$ffa
tosl	$ffd

tosh	$ffe
portc	$f82, bits 3 -7
portc-ddr	$f94, , bits 3 -7

All other special file registers may be written to without interfering with the operation of the LogoChip. Please see the PIC2320 data sheet published by Microchip for information about the functionality of these registers.

General Purpose Registers (RAM)

The PIC18F2320 microcontroller has 512 bytes of general purpose memory (RAM). A map of how this RAM is used by the Logo virtual machine is shown in the figure below. Note that some of this memory is used by the Logo virtual machine and should not be written to by Logo code.

RAM Address	Use
$00 - $1f	Used by Logo Virtual Machine - Do Not Overwrite
$20 - $ff	Global variables stored here
$100 - $1bf	Used by Logo Virtual Machine (for Logo stack) - Do Not Overwrite
$1c0 -$1ff	Transfer buffer used during download only

Flash Program Memory Map

The PIC18F2320 has 8k bytes EEPROM (electrically erasable programmable read only memory.[4] The addresses of these memory locations are $00-$1fff. In the LogoChip this memory is allocated as follows:

Locations $0000 - $01ff (0-511) Boot Monitor – These locations are used for the “monitor” which is responsible for writing to the microcontroller’s program memory. When the LogoChip is started in “bootload mode”(by holding the start/stop button down) the monitor is responsible for programming new assembly language programs (such as the Logo virtual machine) into the memory space between $0200 and $0bff. When not started in bootload mode the execution simply jumps to location $0200, where the Logo virtual machine begins. The Logo virtual machine uses the monitor to write new user Logo program bytes codes.

[4] Microchip specifies that this memory will allow at least 100,000 erasures.

Locations $0200 - $0bff - (512- 3071) Logo Virtual Machine – These locations are for the LogoChip Virtual Machine. Assembly language programs can also be written into this space, overwriting the virtual machine and allowing the user to run ordinary PIC machine code, while still maintaining the monitor.

Locations $0c00 - $0c3f - (3072-3135) Command Center Storage - Logo byte codes generated from programs that are run from the command center are stored here.

Locations $0c40 - $0c7f - (3136- 3199) Startup and Powerup Vectors - contains the address of the program memory location for the Logo procedure to be run the `startup` and `powerup` procedures.

Locations $0c80 - $0cff - (3200-3327) - are unused by the LogoChip.

Locations $d00 - $1fff (3328 - 8191) - User Logo Program Byte Codes - The Logo programs that a user writes get translated into a series of one-byte codes, which get stored in this section of memory during the downloading process. Because this memory is non-volatile, user programs remain in the LogoChip when the power is turned off. They are overwritten with each new download.

Table of LogoChip Logo Byte-Codes

Mnemonic	Byte	Command or Reporter	Number of Args	Comment
code-end	0	c	0	terminates code.
byte	1	r	1 immediate byte	pushes a 16-bit number on stack. takes one immediate byte as number in code stream (resulting number is 16-bit representation of 0 - 255).
number	2	2	2 immediate bytes	pushes 16-bit number on stack. takes two immediate bytes as number in code stream.
list	3	c	0	"start of list"; opens a code block
eol	4	c	0	"end of list"; closes a code block
eolr	5	r	0	"end of list reporter"; closes a code block that will return a value (*e.g*, for when, waituntil)
lthing	6	r	1	"local thing"; uses stack frame to retrieve procedure arguments
ufun	7	n/a	2 immediate bytes	call a user function
eval-ufun-tail	8	c	2 immediate bytes	tail recursively call the same procedure
stop	9	c	0	stops currently running procedure, returning control to caller
output	10	c	1	stops currently running procedure, returning value to caller
loop	11	c	1	indefinitely executes block
repeat	12	c	2	repeats block for specific number of times
if	13	c	2	if input expression is true, executes block
ifelse	14	c	3	if input expression is true, executes block1, else executes block2
waituntil	15	c	1	repeatedly executes block until it evaluates to true
+	16	r	2	reports sum of two inputs
-	17	r	2	reports difference of two inputs
*	18	r	2	reports product of two inputs
/	19	r	2	reports quotient of two inputs
%	20	r	2	reports remainder of quotient of two inputs
=	21	r	2	reports boolean equality of two inputs
>	22	r	2	reports boolean "greater than" of two inputs
<	23	r	2	reports boolean "less than" of two inputs

and	24	r	2	reports bitwise/logical AND of two inputs
or	25	r	2	reports bitwise/logical OR of two inputs
xor	26	r	2	reports bitwise/logical XOR of two inputs
not	27	r	1	reports bitwise/logical NOT of two inputs
read	28	r	1	reports 8-bit value of file register at specified 8-bit address
write	29	c	2	writes 8-bit value to file register at specified 8-bit address
global	30	r	1	reports value of numbered global
setglobal	31	c	2	sets numbered global to value
resett	32	c	0	resets free-running timer to zero
timer	33	r	0	returns value of free-running timer
wait	34	c	1	waits for specified time period
random	35	r	0	reports pseudorandom 16 bit value
send	36	c	1	sends specified 8-bit value out LogoChip serial port
lowbyte	37	r	1	reports low byte of input
highbyte	38	r	1	reports high byte of input
setbit	39	c	2	sets a specified bit in a specified file register
clearbit	40	c	2	clears a specified bit in a specified file register
togglebit	41	c	2	toggles a specified bit in a specified file register
testbit	42	r	2	reports the state of a specified bit in a specified file register
leftshift	43	c	2	shifts to the left the contents of a specified register by a specified number of spaces
read-rom	44	r	1	reports 14-bit value of program memory at specified 16-bit address
no-op	45	c	0	does nothing (takes about 13 microseconds)
flash	46	c	0	causes a “bootflash” to occur on the indicator LED
read-ad	47	r	1	reports the results a 10-bit analog to digital conversion on a specified channel (0-3, 5)
print	48	c	1	prints a 16-bit value in the monitor box on the right side of the LogoChip Logo screen and moves cursor to the next line
prs	49	c	1	prints the character string that follows the quotation mark in the monitor box on the right side of the LogoChip Logo screen and moves cursor to the next line?
mwait	50	c	1	wait for a specified number of milliseconds
stop!	51	c	0	terminates execution completely.

Notes

* - A "command" does not return a value; a "reporter" returns a single value on the stack.

* - Most arguments are taken from the execution stack. Exceptions are "byte" and "number", which push constants from the bytestream onto the stack, and opcodes that take code blocks (e.g., if, ifelse, when); code blocks are inline the code stream.