The Handy Logo Language Reference

Fred Martin and Brian Silverman
MIT Media Laboratory*

January 12, 1996

The Handy Board is a hand-held, battery-powered computer that can receive inputs from electronic sensors (including touch, light, and sound sensors) and operate DC motors. It is designed for a variety of educational robotics uses, including mobile robot projects, data-taking applications, and "ubiquitous computing" applications (projects that embed computers in the world around us).

This document explains how to use the Handy Logo programming language, assuming the Handy Logo version dated January 12, 1996.

*20 Ames Street Room E15–320, Cambridge, MA 02139. The Handy Board hardware was designed primarily by Fred Martin; the Handy Logo software system was designed primarily by Brian Silverman. To contact the authors: `fredm@media.mit.edu` and `bss@media.mit.edu`

Contents

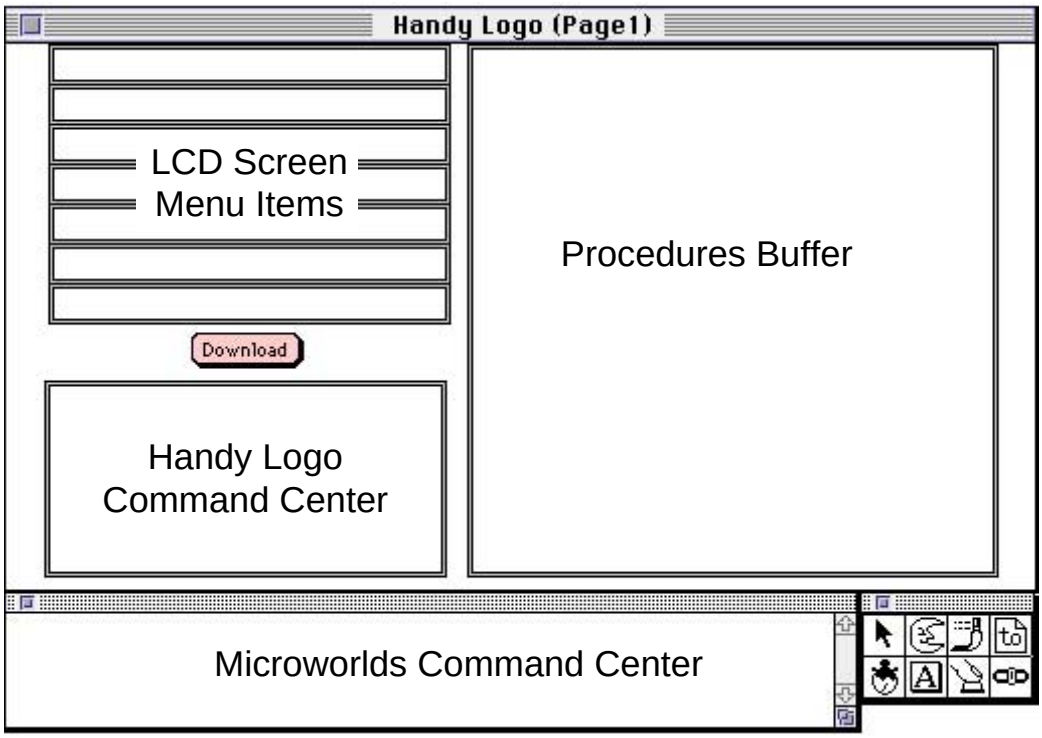

Figure 1: Handy Logo Screen Shot

1 Handy Logo Interface

Figure 1 shows a screen shot of the Handy Logo interface. Here is an explanation of each of the screen windows:

Procedures Buffer. Handy Logo procedures are typed here.

LCD Screen Items. Logo statements to be displayed on the Handy Board's LCD screen are typed here.

Handy Logo Command Center. Logo statements to be run immediately are typed here. When the return key is pressed, the line just typed is compiled and downloaded to the Handy Board for immediate execution.

Download Button. To compile new procedures and/or LCD screen items, click on the Download button. All procedures and screen items are then compiled

and downloaded to the Handy Board.

Microworlds Command Center. Instructions to be given to the underlying Microworlds Logo interpreter are typed here.

2 Motor Outputs

Motors are labelled on the Handy Board itself as *Motor-0*, *Motor-1*, *Motor-2*, and *Motor-3*, but are referred to in Handy Logo by letter, with "a" being *Motor-0*, "b" being *Motor-1*, etc.

Each motor output is capable of bidirectional control of a DC motor, at eight steps of power from off to full on.

`a,` Selects motor A for subsequent commands.

`b,` Selects motor B.

`c,` Selects motor C.

`d,` Selects motor D.

`ab,` Selects motors A and B together.

`bc,` Selects motors B and C.

`ac,` Selects motors A and C.

`ad,` Selects motors A and D.

`abcd,` Selects all motors.

`on` Turns selected motor(s) on.

`off` Turns selected motor(s) off.

`toggle` Inverts on/off state of selected motor(s); i.e., motors that are off go on, and motors that are on go off.

`rd` Reverses direction of selected motor(s).

`thisway` Sets selected motor(s) for one of the two possible directions (indicated by the green motor LED being illuminated). When motors are first turned on, they are in the "thisway" state.

`thatway` Sets selected motor(s) for the other of the two directions (indicated by the red motor LED being illuminated).

`onfor` *time* Turns selected motor(s) on for *time* tenths of seconds.

`setpower` *level* Sets the power level of the selected motor(s). Power levels range from 8 (full power) to 0 (off). The initial state of motors, when turned on, is full power.

3 Sensor Inputs

The Handy Board has two sensor banks: the analog inputs, numbered 0 to 6 on the board, and the digital inputs, numbered 7 to 15 on the board.

`switch` *number* Reports value of switch sensor *number* plugged into digital or analog input bank. If the switch is closed, the value "true" is reported.

`sensor` *number* Reports value of sensor *number* from analog sensor bank as a value from 0 to 255.

`timer` Reports amount of elapsed time in milliseconds. Reset by `resett` or pressing STOP button.

`resett` Resets elapsed time count to zero.

4 Control Structures

`wait` *time* Waits (does nothing) for *time* tenths of seconds.

`waituntil [` *condition* `]` Waits until *condition* becomes true.
Example: `waituntil [sensora > 180]`

`if` *condition* `[` *action* `]` Performs *action* if *condition* is true. Typically used in a loop to repeatedly test the condition. Example: `if switcha [ab, rd]`

`ifelse` *condition* `[` *action* `] [` *else-action* `]` Performs *action* if *condition* is true; otherwise, performs *else-action*. Example: `ifelse sensora > 180 [ab, on][ab, off]`

`repeat` *times* `[` *action* `]` Repeatedly performs *action* for *times* number of times. Example: `repeat 10 [ab, onfor 10 rd]`

`loop [` *action* `]` Indefinitely loops performing *action*. To exit from within this loop, use the `stop` command, which causes currently running procedure to terminate.

5 User Input/Output

5.1 Printing

The Handy Board can print information to its LCD screen as well as over the serial line to be displayed on the host computer console.

`print "`*word* Prints a single word to the LCD screen. Example: `print "hello`

`print [` *word1 word2 word3* . . . `]` Prints phrase to the LCD screen. Example: `print [hello there matey]`

`print` *number* Prints a number to the LCD screen. Example: `print sensora`

`type` Used like `print`, but allows multiple statements to print onto the same display line. Example: `type [Sensor is] print sensora`

`top` Selects top line of LCD screen for subsequent printing.

`bottom` Selects bottom line of screen.

`say "`*word* Outputs a single word over the serial line for display on the host computer console. Example: `say "hello`

`say [` *word1 word2 word3* . . . `]` Outputs phrase to the serial line. Example: `say [hello there matey]`

`say` *number* Outputs a number to the serial line.

5.2 Input

The following describes the action of the Start and Stop buttons.

START button. Pressing the START button causes the screen item currently displayed on the Handy Board's LCD screen to be run (if it was idle). An asterisk is displayed in the lower right corner of the screen while the item is running. If the screen item was already active when the START button is pressed, then the item's process is stopped.

STOP button. Pressing the STOP button causes all processes running on the Brick to be stopped. All motor outputs are turned off. Additionally, the internal motor state is reset to the power-on defaults: all motors at `setpower 8`, direction `thisway`, and "talking to" Motor A.

5.3 Sound

`beep` Plays a short beep on the Handy Board's beeper.

`note` *midi-step duration* Plays a specific tone on the Handy Board's beeper. Pitch is determined by *midi-step* number, which represents successive semi-tones as value increases. Audible values range from about 40 (low tones) to 120 (high tones). *duration* is specified in tenths of seconds.

5.4 Infrared Communication

The Handy Board receives infrared commands from a Sony-brand infrared remote (or a universal remote programmed to transmit Sony codes). Keys 1 through 7 cause the first through seventh screen item, respectively, to be run.

When the Handy Board is running a program, the POWER key will cause the program to stop (this is equivalent to pressing the STOP button). In addition, Handy Logo programs can use the following primitives to send and receive infrared codes. Note that if a Handy Board transmits the code corresponding to the "1" key to another Handy Board, the Handy Board receiving the transmission will run the screen item corresponding to the key. If this program is already running, receiving the code will stop execution; otherwise, it will initiate it.

`ir` Reports a number corresponding to a key on an infrared remote or signal transmitted from another Handy Board.

`irsend` *value* Sends *value* from 0 to 255 to another Handy Board, using infrared transmitter accessory plugged into the infrared output port. Note translation table below.[1]

Transmitted Character	*Received Action*
128 or 18	runs/stops menu item 1
129 or 20	runs/stops menu item 2
130 or 19	runs/stops menu item 3
131 or 17	runs/stops menu item 4
132	runs/stops menu item 5
133	runs/stops menu item 6
134	runs/stops menu item 7
149 or 223	stops all processes & motors

`remote-off` Disables the Handy Board from automatically running menu items based on receiving infrared codes. Important when using the Handy Board out of doors or under direct illumination from halogen light, since these light sources cause spurious information, such as the Stop code, over the infrared input sensor.

6 Serial Line

The Handy Board can send characters over the serial line while it is executing Handy Logo programs. The serial line communications setting is 9600 baud, eight bit data, no parity.

`send` *char* Transmits lower byte of *char* over serial line.

7 Multi-Tasking

The Handy Board can support up to sixteen concurrent process tasks. Each of the following primitives launches a new task.

[1] This table is used to translate the channel/volume up/down keys, from a Casio infrared watch, into the codes for buttons 1 through 4. The 149 code is the Power key, and the 223 code is the Stop key on Sony CD player remotes.

7.1 Starting Processes

`launch [` *action* `]` Launches *action* as a separate process, and starts a new process family (see Section 7.2, below).

`forever [` *action* `]` Launches a process to repeatedly execute *action*.

`when [` *condition* `][` *action* `]` Launches a process to repeatedly test *condition* and execute *action* when it becomes true.

The condition clause for the `when` statement fires on edge-triggered logic; that is, *action* is run once for each time the *condition* changes from false to true. In the case in which the *condition* is true the first time the `when` statement is executed, the *action* is not run.

`every` *time* `[` *action* `]` Launches a process to execute *action* every *time* tenths-of-seconds.

7.2 Stopping Processes

Pressing the STOP button or sending the infrared stop code stops all running tasks, turns off motors, and resets the internal motor state (see 5.2).

Additionally, the "stoprules" primitive may be used for process control. `stoprules` kills all processes begun in the current process family. A new process family is formed every time the `launch` primitive occurs, when a command is started in the command center, and when a command is started from a menu item. `stoprules` does *not* terminate execution of the procedure that calls it.

Here are two examples. In the first, a procedure launches an `every` process ("mode 1"), waits for a trigger condition, uses `stoprules` to halt this, and then launches a new `every` process ("mode 2"):

```
to mode1thenmode2
  a, every 10 [onfor 2] ; mode 1
  waituntil [switch 0] ; trigger for mode 2
  stoprules
  b, every 10 [onfor 2] ; mode 2
end
```

The `mode1thenmode2` procedure consists of one launch family; after the `stoprules` command, mode 1 is halted and mode 2 is launched.

The second example is more sophisticated because it demonstrates how modes may be both stopped and restarted. The `main` procedure launches two process families (`mode1-control` and `mode2-control`) which take care of starting and stopping themselves based on different conditions:

```
to main
  launch [mode1-control]
  launch [mode2-control]
end

to mode1-control
  loop [waituntil [starting-condition]
        mode1 ; begin mode1 tasks
        waituntil [stopping-condition]
        stoprules ; stop mode1 tasks
  ]
end

to mode2-control
  (same form as mode1-control)
end

to mode1
  (mode1 when's, every's, and forever's)
end

to mode2
  (mode2 when's, every's, and forever's)
end
```

In the example, the two modes are responsible for starting and stopping themselves (though this happens asynchronously). If it were desirable to have a third process control the other two, this should be done by having the third process set global variables that the other processes examine to enable and disable their activity (see comments at Section 7.3).

`stoprules` Stops all processes in the current process family. Has no effect when issued from the command center.

7.3 Process Granularity

Handy Logo allows each process to evaluate *one Logo statement* per time slice. A Logo statement is defined as a piece of Logo code that does not return a value. For example, each of the following is an individual Logo statement and will fully execute in one time slice:

```
on
onfor 10
onfor 3 + sensor 5
setfoo digital 3 (assumes foo is declared as a global)
if not foo [setfoo 1] (assumes foo is declared as a global)
```

As indicated by the final example, this time-slicing protocol allows Handy Logo to support semaphors for dynamic process interaction. The semaphor is typically used to coordinate shared control of a resource. A global variable indicates whether the resource is free; if it is, a process then sets the global to indicate that it has taken control of the resource. It is necessary for this "test and set" operation—the semaphor—to happen in a single time slice, so that a process does not perform the test, get swapped out, and then return to take control of a resource that has been grabbed in its absence. The final example shows how the test-and-set operation should be performed in Handy Logo.

8 Data Recording and Playback

There is a single global array for storing data which holds 16,382 2-byte integer values. There is no error-checking to prevent against overrunning the data buffer.

`erase` Resets the data recording pointer to zero.

`record` *value* Records *value* in the data buffer, and advances the recording pointer.

`record#` Reports value of record pointer, indicating where the next data point to be recorded will go.

`resetr` Resets the recall pointer to zero.

`recall` Reports value of current data point, and advances the recall pointer.

`recall#` Reports value of recall pointer.

9 Procedures, Variables, and Comments

9.1 Procedure Definition

Procedures are defined using the keyword “to”; i.e.:

```
to test
  procedure body
end
```

9.2 Procedure Inputs

Inputs, or arguments, to procedures are declared using the standard Logo colon syntax; e.g.:

```
to test :input1 :input2
  top type [Input 1 is] print :input1
  bottom type [Input 2 is] print :input2
  wait 10
end
```

Procedure inputs are local variables.

9.3 Local Variables

Local variables are declared using the `let` keyword, accessed using Logo’s colon syntax, and set using the `make` keyword:

```
to local-example
  let [alocal 5 anotherlocal 17]
  print :alocal ; prints "5"
```

```
  make "anotherlocal 3
  print :anotherlocal ; prints "3"
end
```

The “let” declaration should be made at the beginning of a procedure.

9.4 Global Variables

Global variables are declared using the `global` keyword, which takes a list of the names of globals to be created; i.e.:

`global [`*name1 name2 name3 ...*`]`

This declaration should come at the beginning of the procedure buffer. After being declared, each global is set using a mechanism in which the global name is preceded by the word “set.” Global values are accessed by using the global name as a reporter; e.g.:

```
global [myglobal]

to test
  setmyglobal 3
  print myglobal
  wait 10
end
```

Global variables maintain their value when the Handy Board is power-cycled.

9.5 Procedure Return Values

By default, procedures do not produce return values. Procedures may return a numeric value using the `output` primitive; e.g.:

```
to double :n
  output :n * 2
end
```

Procedures may terminate at any point using the `stop` primitive, which exits the procedure without producing a return value.

Care should be taken to ensure that a procedure either *always* or *never* exits with a return value.

9.6 Code Comments

There are two forms for comments in the procedure buffer:

- Any text between the `end` statement of one procedure and the `to` declaration of the next procedure is ignored.
- Any text after a semicolon (";") on any given line is ignored.

10 Numeric Operations

Handy Logo is based on signed 16-bit integer arithmetic (all numeric values are in the inclusive range from -32768 to $+32767$).

All of the following arithmetic and boolean operators must be preceded and followed by a space. For example, the following expression is *not* legitimate:

```
print 3+4 ; this does not work
```

10.1 Arithmetic Operators

The following arithmetic operators are supported, using infix notation:

`+` — addition.

`-` — subtraction.

`*` — multiplication.

`/` — division.

`\` — remainder.

The minus sign may also be used as a prefix negation operator.

10.2 Boolean and Bitwise Operators

The Boolean operators always produce values of zero or one. In evaluating conditionals, zero is false; any value other than zero is true.

`and` — performs bitwise "and" function. Prefix.

`or` — performs bitwise "or" function. Prefix.

`not` — performs Boolean logical negation. Prefix.

`>` — performs Boolean test for greater-than. Infix.

`<` — performs Boolean test for less-than. Infix.

`=` — performs Boolean test for equality. Infix.

Since the Boolean operators produce values of one and zero, and non-zero results are considered true, the `and` and `or` operators, which are bitwise, can function as Booleans when combining the result of other conditionals. The following example illustrates correct usage:

```
if and (:value > 100) (:value < 150) [doit]
```

10.3 Precedence

Order of evaluation is from left to right; standard rules of precedence are *not* observed. Parentheses may be used to override the standard order of evaluation.

10.4 Random Numbers

Handy Logo includes a primitive for generating pseudo-random numbers. The "random" primitive takes as input the upper limit of the number to be generated, and reports a value between 0 and that number minus 1 (inclusive).

Random's operation is based on a 2 MHz system clock inside the 6811 CPU. When random is called, the clock is sampled and the modulus of the low bits and the input to random is returned.

`random` *limit* Reports a pseudo-random number between 0 and *limit* − 1 (inclusive).

11 Memory Access

Four primitives exist for directly accessing system memory.

`eb` *address* Examine Byte. Reports the byte located at memory *address* as a value from 0 to 255.

`ew` *address* Examine Word. Reports the 16-bit value located at memory *address*.

`db` *address value* Deposit Byte. Stores the lower byte of *value* at memory *address*.

`dw` *address value* Deposit Word. Stores *value* at memory *address*.

12 File Management

To save and load Handy Logo programs, please use the following (rather than saving multiple copies of the Handy Logo project file):

`saveall "`*filename* Saves procedures and screen items into file named *filename*.

`loadall "`*filename* Loads procedures and screen items from file named *filename*.

These commands must be typed into the Microworlds command center, located at the bottom of the computer screen, not the Handy Board command center.