E   Brick Logo Quick Reference

Brick Logo is the language used to write programs that run on the Brick. Brick Logo is similar to the versions of Logo that are part of the commercial LEGO Dacta products (both *LEGO tc logo* and *LEGO Control Lab*). Previous experience with either of these two products, as well as any other Logo experience, will translate easily to writing programs for the Programmable Brick.

E.1   Motors

Motors A, B, and C are bi-directional (the motors' can be reversed under software control). Motor D is uni-directional—the Brick can only turn the motor on and off, and the direction is determined by the way the cable is connected.

`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.

`abc,`   Selects motors A, B, and C.

`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.

E.2 Sensors

switcha

switchb

switchc Reports value of switch sensor (pressed is "true," not pressed is "false.")

sensora

sensorb

sensorc

sensord

sensore

sensorf Reports value of sensor as a number from 0 to 255.

countera

counterb

counterc Reports counts on angle sensor.

resetca

resetcb

resetcc Resets count to zero.

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

resett Resets elapsed time count to zero.

battery Reports battery level as a percentage of full charge (0 to 100).

E.3   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 [ad, rd]`

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

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

loop [*action*]   Indefinitely loops performing *action*. To exit, use `stop` com-
mand, which causes currently running procedure to terminate.

E.4   Input/Output

E.4.1   LCD Display

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

print [*word1 word2 word3* ...]   Prints phrase to the computer screen. Ex-
ample: `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 display for subsequent printing.

bottom   Selects bottom line of display.

E.4.2 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 Brick'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 talkto state `a,`.

E.4.3 Sound

`note` *midi-step duration* Plays a tone on the Brick'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.

E.4.4 Infrared Communication

The Brick 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 Brick is running a program, the Power key will cause the program to stop (this is equivalent to pressing the Stop button). In addition, Brick Logo programs can use the following primitives to send and receive infrared codes. Note that if a Brick transmits the code corresponding to the "1" key to another Brick, the Brick 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 Brick.

`irf` Reports number received by infrared sensor plugged into sensor port F.

`irsend` *value* Sends *value* from 0 to 255 to another Brick, using infrared trans-
mitter accessory plugged into motor port D. Note translation table below.[5]

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

E.4.5 Serial Line

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

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

E.4.6 Speech Output

The Brick can connect to a specially-modified version of RC Systems' voice board
for natural-speech output.[6] The "say" primitive is used to transmit information to
the voice board over the Brick's serial line connection:

`say "`*word* Outputs a single word to the voice board. Example: `say "hello`

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

---

[5]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.

[6]Contact the authors for information about how to wire the voice board to the Brick.

say *number* Outputs a number to the voice board. For example, `say sensora` would result in the current value of sensor A being transmitted. The voice board converts the numeric representation (e.g., "193") to its spoken form (e.g., "one hundred ninety three").

After any power-on, it is necessary to send the voice board an odd-numbered-byte over the serial line, followed by a short delay, to establish communications baud rate. The carriage return character, 13, is a good choice. Also, it is necessary to send the carriage return to get the board to speak words that have been already transmitted:

```
to init-speech-board
  send 13 wait 1
end

to test-speech-board
  say [hello there.]  send 13
end
```

E.5   Multi-Tasking

The Brick can support up to eight concurrent process tasks. Each of the following primitives launches a new task.

E.5.1   Launching Processes

`launch [` *action* `]` Launches *action* as a separate process.

`forever [` *action* `]` Launches a process to repeatedly execute *action*. Equivalent to `launch [loop [` *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 each time that *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.

E.5.2 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 E.4.2).

`stoprules` Stops all processes except the one executing the "stoprules" command.

E.6 Data Recording and Playback

There is a single global array for storing data which holds 5887 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.

E.7 Procedures, Variables, and Comments

E.7.1 Procedure Definition

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

```
to test
  procedure body
end
```

E.7.2 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.

E.7.3 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.

E.7.4 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"; their 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 Brick is power-cycled.

E.7.5 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.

E.7.6 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.

E.8   Numeric Operations

Brick 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
```

E.8.1   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.

E.8.2   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]
```

E.8.3 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.

E.9 File Management

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

`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 Brick command center.

It is also possible to save an entire Brick Logo project (procedure definitions and screen items) to a Brick. Use the following commands:

`savetobrick` Saves procedures and screen items to a Brick.

`loadfrombrick` Loads procedures and screen items from a Brick.