Handy Cricket Programming Reference


Overview

The Handy Cricket is programmed in a language called "Cricket Logo," which is a simplified version of the powerful yet easy-to-learn Logo language.

Cricket Logo has the following features:

- procedure definition with inputs and return values;
- global variables and local procedure inputs;
- control structures like `if`, `repeat`, and `loop`;
- a 16-bit number system (addition, subtraction, multiplication, division, remainder, comparison, bitwise operations, random function);
- motor and sensor primitives;
- timing functions and tone-playing functions;
- data recording and playback primitives;
- communications primitives.

Cricket Logo Screen
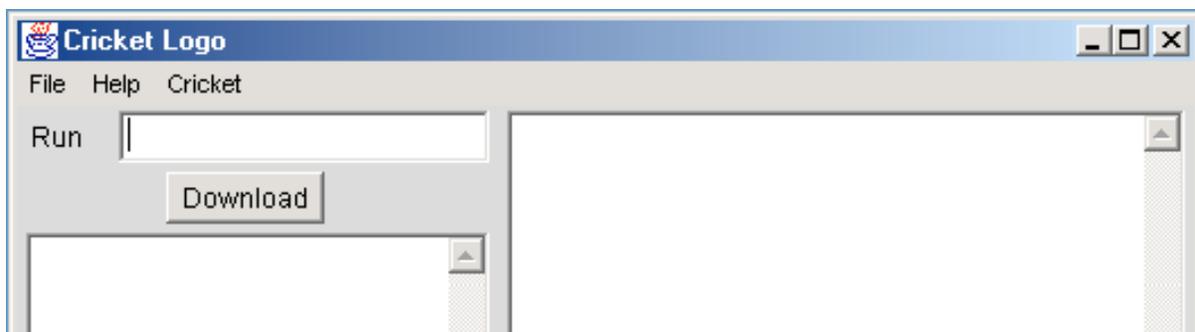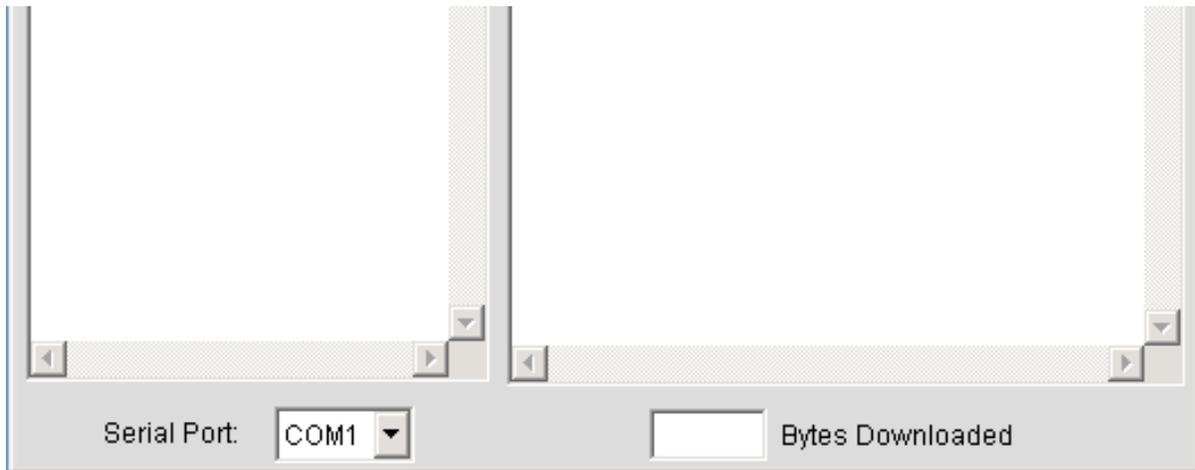
The Cricket Logo screen is shown above.

The text area in the lower left is the *command center*. Statements typed into this window are immediately downloaded to the Cricket and then executed. In the screen image, the primitive `beep` is shown. This command causes the Cricket to emit a short chirp.

The text area on the right is for procedures, labeled as such at the bottom of the window. In Cricket Logo, procedures are defined using `to` and `end` syntax. For example, the following procedure, named `demo`, turns on motor A for one second and then beeps:

```
to demo
   a, onfor 10
   beep
end
```

To download procedures, click the "Download" button. This causes all procedures written in the procedures area to be downloaded to the Cricket.

The one-line text area at the top left of the window is the Run Line. After program download, any statement typed into this area will be run when the Cricket's Run/Stop button is pressed.

Motors

The Cricket has two motors, which are named "A" and "B." A bi-color LED indicates the state of each motor.

Motor commands are used by first selecting the motor (using `a,` `b,` or `ab,`) and then telling it what to do (e.g., `on`, `off`, `rd`, etc.).

`a,`	Selects motor A to be controlled.
`b,`	Selects motor B to be controlled

`ab,`	Selects both motors to be controlled.
`on`	Turns the selected motors on.
`off`	Turns the selected motors off.
`onfor` *duration*	Turns the selected motors on for a *duration* of time, where *duration* is given in tenths-of-seconds. E.g., `onfor 10` turns the selected motors on for one second.
`thisway`	Sets the selected motors to go the "thisway" direction, which is defined as the way that makes the indicator LEDs light up green. `thisway` is the default motor direction when the Cricket is first powered on.
`thatway`	Sets the selected motors to go the "thatway" direction, which is defined as the way that makes the indicator LEDs light up red.
`rd`	Reverses the direction of the selected motors. Whichever way they were going, they will go the opposite way.
`setpower` *level*	Sets the selected motor(s) power level. Input is in the range of 0 (coasting with no power) to 8 (full power). The Cricket motor outputs start out at power level 4.

Timing and Sound

The timing and sound primitives are useful to cause the Cricket to do something for a length of time.

Timing

With the `wait` primitive, one can cause the Cricket to "do nothing," thereby causing for a certain amount of time to pass by. While the Cricket is idling in this fashion, motors may be left on, so, for example:

```
ab, on wait 20 off
```

will turn the motors on for two seconds. This is equivalent to:

```
ab, onfor 20
```

There is also a free-running timer, which keeps track of elapsed time even when the Cricket is doing other things. Two primitives are available for using the timer: `resett`, which resets the timer to zero, and `timer`, which reports the current value of the timer.

The table below summarizes the `wait`, `timer`, and `resett` primitives.

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

| `timer` | Reports value of free-running elapsed timer. Time units are reported in milliseconds. So if `timer = 1000`, that would indicate 1 second of elapsed time. Please note: the timer is only updated every 4 milliseconds, so if you were watching it continuously, you would see it count 0, 4, 8, 12, ... etc. |
| `resett` | Resets the elapsed time counter to zero. |

Sound

The Cricket has a built-in piezo beeper which may be used to play simple tones. There are two primitives for making sound: `beep`, which generates a short beep of fixed length and pitch, and `note`, which takes duration and pitch inputs:

| `beep` | Plays a short beep. |
| `note` *`pitch  duration`* | Plays a note of a specified pitch and duration. Increasing values of the *pitch* create lower tones (the pitch value is used as a delay counter to generate each half of the tone's squarewave). The *duration* value is specified in tenths-of-seconds units. The correspondence between the numbers used to define the pitch and the musical notes in the octave between middle c and high c is shown in the table below. |

pitch number	119	110	110	105	100	100	94	89	84	84	79	74	74	70	66	66	62	59
musical notation	c	c#	db	d	d#	eb	e	f	f#	gb	g	g#	ab	a	a#	bb	b	c2

For example, `note 119 5` will play a middle c for half a second.

Please note that there are two different reference values for timing: 0.1 second units, used in `wait`, `onfor`, and `note`, and 0.004 second units, used in `timer`.

Sensors

The Cricket has two sensor ports, named "A" and "B." Various devices may be plugged into these sensor ports, including:

- on/off devices like pushbuttons or lever switches;
- variable-resistance devices such as light-sensitive photocells and temperature-sensitive thermistors;
- any other electronic circuit that generates a voltage between 0 and 5 volts.

There are two types of primitive for reporting the value of a sensor plugged into a sensor port: *switch*, which reports a true-or-false reading (assuming a switch-type sensor), and *sensor*, which reports a numerical reading (from 0 to 255) depending on the sensor's value:

`switcha`	Reports true if the switch plugged into sensor A is pressed, and false if not.
`switchb`	Reports true if the switch plugged into sensor B is pressed, and false if not.
`sensora`	Reports the value of sensor A, as a number from 0 to 255
`sensorb`	Reports the value of sensor B, as a number from 0 to 255.

Control Structures

Cricket Logo supports a small but useful set of control forms. These consist of loop structures, conditionals, a busy-wait form, and primitives for early termination of procedure execution.

Overview

The following table summarizes Cricket Logo's control structures:

`repeat times [body]`	Executes *body* for *times* repetitions. *times* may be a constant or calculated value.
`loop [body]`	Repetitively executes *body* indefinitely.
`if condition [body]`	If *condition* is true, executes *body*. A conditional expression that evaluates to zero is considered false; 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.
`stop!`	Fully terminates execution; procedure does *not* return to its caller. Note: if there is a background task running (see `when`), it will *not* be stopped by the `stop!` command (use `whenoff` to terminate the background task).
`output value`	Terminates execution of procedure, reporting *value* as result to calling procedure.

Examples

The following procedure makes the motor A output flip back and forth ten times:

```
to flippy
   repeat 10 [a, onfor 10 rd]
end
```

Here are two ways to make the motor A flip back and forth continuously:

```
to flippy-forever-1
   loop [a, onfor 10 rd]
end

to flippy-forever-2
   a, onfor 10 rd
    flippy-forever-2
end
```

The second method, which avoids the use of the `loop` form, is more efficient, because the compiler recognizes it as tail recursion and optimizes it as a goto.

The following procedure turns on motor A, waits for switch B to be pressed, and then turns off the motor:

```
to on-wait-off
   a, on
   waituntil [switchb]
   off
end
```

The following procedure repeatedly tests the state of switch B. If it's pressed, the motor is made to go on in the "thisway" direction. If the switch is not pressed, the motor goes in the opposite direction:

```
to switch-controls-direction
   a, on
   loop [
      ifelse switchb [thisway][thatway]
   ]
end
```

Numbers

The Cricket has a 16-bit number system. Numbers may range from -32768 to +32767.

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

Normal expression precedence rules are *not* used by the compiler. Instead, operators are evaluated in the order they are encountered from left to right. Thus:

```
3 + 4 * 5
```

evaluates to 35, because the compiler first acts on the 3 + 4 and then multiplies this sum by 5.

Parentheses may be liberally applied to get expressions to evaluate how you intend them. E.g.:

```
(3 + (4 * 5))
```

yields 23.

The following table lists the operators provided in Cricket Logo:

+	Infix addition.
–	Infix subtraction.
*	Infix multiplication
/	Infix division.
%	Infix modulus (remainder after integer division).
and	Infix AND operation (bitwise).
or	Infix OR operation (bitwise).
xor	Infix exclusive-OR operation (bitwise).
not	prefix NOT operation (logical inversion, not bitwise).
random	Reports pseudo-random number from -32768 to +32768. Use the modulus operator to reduce the range; e.g., `(random % 100)` yields a number from 0 to 99.

Procedures, Inputs, and Outputs

Definition

A procedure is defined using the `to` keyword, followed by the procedure name, followed by the procedure body, followed by the `end` keyword. For instance, the following defines the procedure `flash`, which turns the motor A output on and off 10 times:

```
to flash
   repeat 10 [a, onfor 5 wait 5]
end
```

Inputs

Procedures can be defined to accept inputs, which then become local variables inside the procedure. This is done using the colon character ":" as shown below. In this example, the `flash` procedure is given one input, which is then used as the counter in the `repeat` loop:

```
to flash :n
  repeat :n [a, onfor 5 wait 5]
end
```

Then one can use this procedure with (e.g.) `flash 5`, `flash 10`, `flash 20`, etc.

There is no fundamental limit to how many inputs a single procedure may have, but, in practice, available memory to hold the procedure's inputs during procedure invocation imposes a restriction.

Outputs

Procedures may return values with the `output` primitive. In the following example, the `detect` procedure returns a value of 0, 1, or 2 depending on the value of sensor A:

```
global [temp]

to detect
  settemp sensora
  if temp < 30 [output 1]
  if temp < 50 [output 2]
  output 3
end
```

The global variable `temp` is defined. In the `detect`, a reading of sensor A is loaded into `temp`. If the reading is less than 30, then a 1 is returned. If the reading not less than 30, then the next test executes, and if the reading is then less than 50, a 2 is returned. If this test fails, then a 3 is returned.

Care should be taken to ensure that if a procedure sometimes produces an output, that it always does. In a procedure that is capable of producing an output, the Cricket will crash if an execution path that does not produce an output is travelled.

Global Variables

Global variables are created using the `global [`*variable-list*`]` directive at the beginning of the procedures buffer. E.g.,

```
global [cats dogs]
```

creates two globals, named `cats` and `dogs`. Additionally, two global-setting primitives, `setcats` and `setdogs`, are instantiated. Thus, after the global directive is provided, one can say

```
        setcats 3
```

to set the value of `cats` to 3, and

```
        setcats cats + 1
```

to increment the value of `cats`.

Global variables are stored in RAM, so their contents are lost when the Cricket is turned off. To save data when the Cricket is turned off, please use the data recording and playback functions, or the global array functions. Both of these store data in the Cricket's non-volatile memory.

Global Arrays

Global arrays are created using the `array` primitive at the beginning of the procedures window. For example,

```
        array [clicks 50 clacks 25]
```

creates two arrays, one named `clicks`, which can hold 50 numbers, and another named `clacks`, which can hold 25 numbers.

Elements in the array are accessed using the `aget` primitive and written using the `aset` primitive:

`aget` *name index*	Retrieves the item at position *index* from array *name*.
`aset` *name index value*	Sets the item at position *index* to *value* in array *name*.

For example, `aset clicks 31 1000` sets the 31st element of `clicks` to have a value of 1000, and `send aget clicks 31` causes the value of the 31st element of `clicks` to be transmitted via infrared.

Array values are stored in the Cricket's non-volatile memory, so their contents are preserved even when the Cricket is turned off.

There is no error-checking to prevent arrays from overrunning their boundaries.

Data Recording and Playback

There is a single global array for storing data which holds 2500 numbers.

`resetdp`	Resets the data pointer to the beginning position.
`setdp` *position*	Sets the value of the data pointer to *position*. The beginning position is zero.
`erase` *num*	Erases *num* data points, starting from the beginning of the data area, and then resets data pointer back to the beginning.
`record` *value*	Records *value* in the data buffer and advances the data pointer.
`recall`	Reports the value of the current data point and advances the data pointer.

For example the procedure `take-data` can be used to store data recorded by a sensor once every second:

```
to take-data
   resetdp
   repeat 2500 [record sensora wait 10]
end
```

A separate program called "Logo Graph" is available for uploading, graphing, and analyzing data.

Please note that there is no error checking to prevent against recording too many data points and thereby overrunning the data buffer.

Please note that recorded values are stored as 8-bit values (0 to 255); if you try to record a number greater than 255, only the lower byte (remainder when dividing by 256) will be recorded.

Recursion

The Cricket supports recursion—that is, procedures that are defined in terms of themselves.

For instance, the mathematical definition of the factorial function is that the factorial of *n* is equal to *n* times the factorial of *(n - 1)*, with the factorial of 1 being 1. This idea may be expressed in Cricket Logo as:

```
to fact :n
   if :n = 1 [output 1]
   output n * fact :n - 1
end
```

To try this out, download the procedure and then type into the Cricket's command center:

```
repeat fact 3 [beep wait 1]
```

You should hear six beeps, since the factorial of 3 is 6.

Please note that the memory for keeping track of recursive calls during execution is quite small, and limited to perhaps six calls deep. If the stack overflows, the Cricket halts and beeps five times.

"Tail recursion" is a special case of recursion when the very last instruction at the end of a procedure is the recursive call. The Cricket supports tail recursion, converting the recursive call into a goto statement, and avoiding the stack depth problem just mentioned.

The following procedure illustrates correct use of tail recursion, creating an infinite loop:

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

Multi-Tasking

The Cricket supports a limited form of multi-tasking. In addition to the conventional thread of execution, the Cricket can have one "background task." This task repeatedly checks a condition. When this condition becomes true, the task interrupts foreground activity and processes its special action. When the background task's action finishes, execution picks up where it left off in the foreground activity.

The background task is started up using the when primitive. For example, consider the following line of code:

```
when [switcha] [beep] loop [a, onfor 5 rd]
```

(This line may be typed into the Cricket's command center to test it out.)

In this example, the background task is looking for the condition switcha. When this becomes true (e.g., a switch plugged into sensor port A is pressed), then the action is run, and the Cricket beeps. Concurrently, the Cricket executes an infinite loop, turning motor A on for a 1/2 second and then reversing its direction.

Please note a few details about how this is happening:

- The when statement must precede the infinite loop—if it followed the loop, execution would never get to the when statement!

- The when statement itself only *initiates* the background activity, and then execution flow passes through the statement (in the example, it then continues into the loop statement). As such, the when statement should not itself be placed into a loop—it only needs to be executed once to get started.

- The action associated with the `when` is executed just once for each time the condition *becomes* true. Thus, in the example, if the switch is held down, the Cricket will *not* beep continuously. The action executes exactly once each time the condition goes from false to true. Thus the switch must be released and re-pressed to get the Cricket to beep again.

- Please note that the both the condition and the action for the `when` statement must each be enclosed in square brackets.

- There can be only one background task operating at any given time. If a `when` statement is executed after the background task has already been started, the subsequent `when` task supercedes the earlier one.

- To stop the background task from operating, use the primitive `whenoff`.

Infrared Communication

Overview

Crickets can send infrared signals to each other using the `send` primitive and receive them using the `ir` primitive. The `ir` primitive reports the last value received. A third primitive, `newir?`, reports true when an IR byte has been received but not yet retrieved.

As an example, consider the following pair of procedures. The first procedure, called `sender`, will run on one Cricket, and generate numbers which are sent to a second Cricket. Here, the `sender` procedure randomly sends a 0, 1, or 2:

```
to sender
  send random % 3
  beep
  wait 30
  sender
end
```

The expression `random % 3` produces a 0, 1, or 2, using the remainder-after-division operator. This value is sent using the `send` primitive. Then the procedure beeps and waits 3 seconds before sending a new number.

On a second Cricket, a receiver procedure, `doit`, will receive these numbers and either turn on motor A, motor B, or both motors depending on the value it gets:

```
to doit
  waituntil [newir?]
  if ir = 0 [a, onfor 10]
  if ir = 1 [b, onfor 10]
  if ir = 2 [ab, onfor 10]
  doit
end
```

Caveats

Please note that the Cricket system uses values 128 through 134 for low-level operations between Crickets. So it's best to not deliberately send those values around, because Crickets that are sitting idle (turned on, but not running a program) will interpret those codes and possibly overwrite their memory.

The Button

When the Cricket is idle, pressing its pushbutton causes it to begin executing remote-start line 1 on the Cricket Logo screen.

When the Cricket is running a program, pressing the button causes it to halt.

Back to Cricket Home

System Commands

`low-byte` *val*	Reports the low byte of the argument; equivalent to *val % 256* but significantly faster.
`high-byte` *val*	Reports the high byte of the argument; equivalent to *val / 256* but significantly faster.
`eb` *addr*	Examine Byte – reports the value located at memory location *addr*. On the Handy Cricket v1.5, memory begins at address 0 and ends at address 4095 (decimal).
`db` *addr val*	Deposit Byte – reports the byte *val* into Cricket memory at location *addr*. Warning: it's easy to over-write Cricket program memory using this command, causing software crashes.

Last modified: Thursday, 30-Jan-2003 14:33:08 EST by fredm