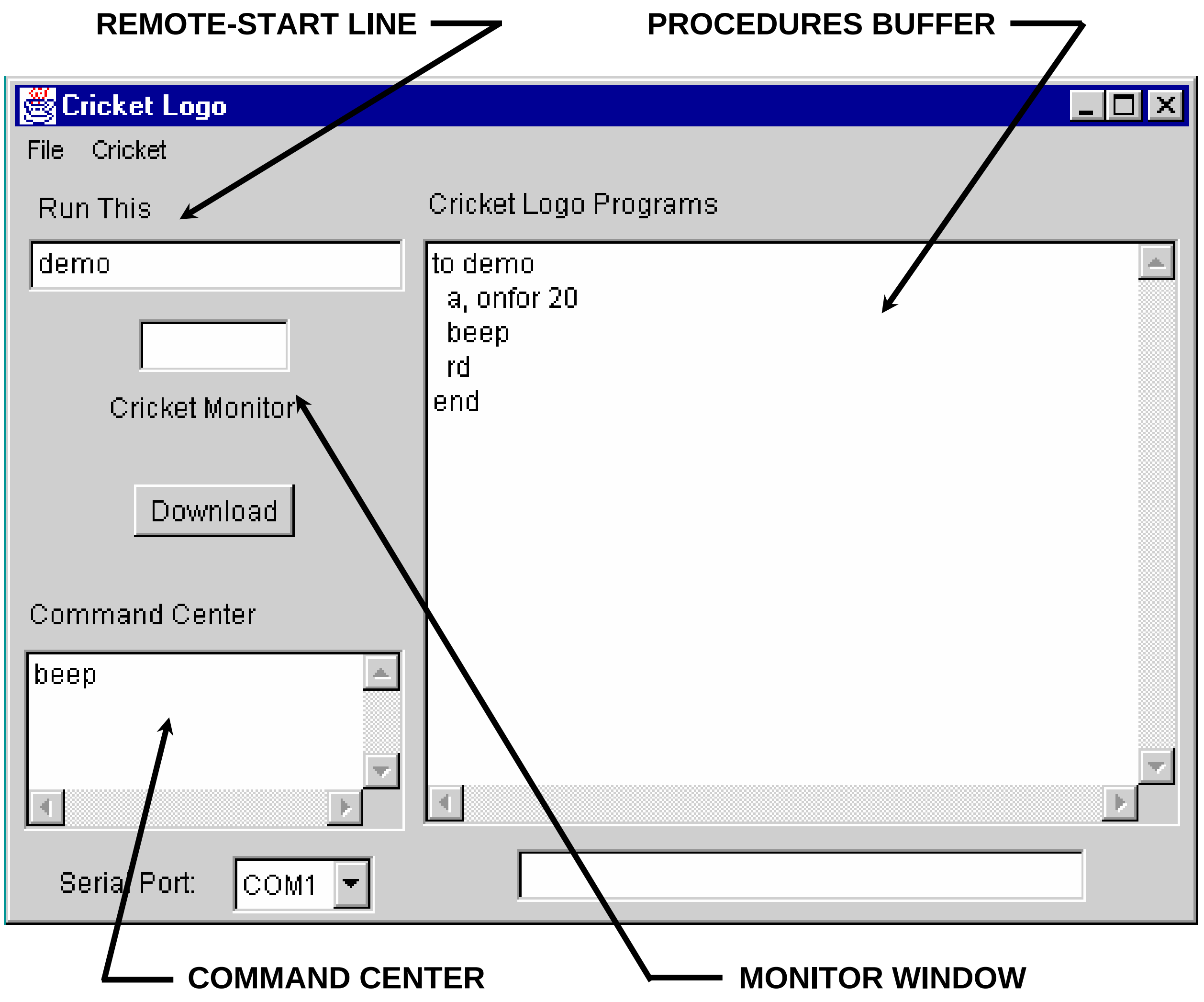
SUPER CRICKET PROGRAMMING GUIDE

Cricket Logo Screen



Run This: demo

```
to demo
  a, onfor 20
  beep
  rd
end
```

Command Center: beep

➢ **<u>The LOWER LEFT text box on the Logo Screen is the COMMAND CENTER</u>.** Statements typed into this window are immediately downloaded to the Cricket and then executed (they happen right away).

- <u>Always start by typing the beep command in the command center</u> to make sure that your interface is communicating properly with the Cricket.

- Plug a motor into port A and a lamp into port B. Type these commands in the command center:

```
a, onfor 30
beep
b, on
setpower 8
setpower 4
b, off
repeat 10 [b, onfor 5 wait 5]
```

Writing Procedures

- **The text box on the RIGHT SIDE of the Logo screen is the PROCEDURES BUFFER** (place for writing programs).

- **All procedures must start with the keyword "to", followed by a one-word procedure name, and the last line must be the keyword "end".**

```
to beepmotor
        a, onfor 20
        repeat 3 [beep wait 5]
        b, onfor 20
        repeat 3 [beep wait 5]
end
```

- **To get your Cricket to run any new or edited procedures, click the "Download" button.** This causes ALL procedures written in the procedures buffer to be sent to the memory of the Cricket microcontroller via the IR interface. **You can run any procedure that has been downloaded to the Cricket by typing the procedure name in the command center.**

  - For above example, type beepmotor in command center & hit return to run it.

- **The Cricket's REMOTE START LINE at the top of the Logo screen tells the Cricket which procedure will be run when the Cricket's Run/Stop button is pressed. If you are downloading more than one procedure, you must enter the name of the procedure that you would like to be "remotely activated" by the run/ stop button.** Any of the procedures, once downloaded, can be run from the command center, however, only the procedure name that was entered into the "Run Line" (at the time of download) can be controlled using the Cricket's run/stop button. **<u>When the Cricket is running a program, pressing its pushbutton causes it to stop</u>**.

- ❑ Try running the beepmotor procedure by pressing the run/stop button on Cricket. It won't work until you actually type beepmotor in the Run Line and then download again. Then every time you press run/stop button, the Cricket will run the beepmotor program.

<u>Output Commands</u>

- ➢ <u>The Cricket has 4 motor or output ports named: a, b, c & d</u>. A motor or any output device (lamp, LED, or relay switch) is controlled by first addressing the output port that is being used and then telling the device what to do (e.g., on, off, reverse direction, etc.).

a, b, c, d,	Selects output port a to be controlled.
ab, cd, abcd,	Selects combinations of ports to be controlled at the same time (cannot use any other combinations)
on	Turns the selected output ports on.
off	Turns the selected output ports off.
onfor 30	Turns the selected output ports on for 3.0 seconds. Time is written in tenths-of-seconds (ex. onfor 12 turns the selected motors on for 1.2 sec).
thisway	Sets the selected motors to spin in the "thisway" direction, which is the direction that makes the indicator LEDs light up GREEN.
thatway	Sets the selected motors to spin in the "thatway" direction, which is the direction that makes the indicator LEDs light up RED.
rd	Reverses the direction of the selected motors – it will now be opposite from the direction they were spinning the last time they were addressed.
setpower 8	Sets the selected output ports to a certain power level, which can range from 0 (no power) to 8 (full power). If no power level is specified, the output port will always default to setpower 4.

- ❑ Plug a motor into port a and a lamp into port b, and try this procedure to control a carnival ride:

```
to accelerate.ride
ab, thisway
setpower 2 onfor 20
setpower 5 onfor 20
setpower 8 onfor 20
wait 20
repeat 5 [beep wait 5]
end
```

Output Commands - Sound

- ➢ The Cricket has a built-in piezo beeper that can play simple tones. There are two primitives for making sound:

beep	Plays a short beep.
note *pitch duration*	Plays a note of a specified *pitch* and *duration*. Increasing values of the *pitch* creates lower tones (just opposite of what you would think). The *duration* value is specified in tenths-of-seconds. 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	105	100	94	89	84	79	74	70	66	62	59
musical notation	c	c#/ db	d	d#/ eb	e	f	f#/ gb	g	g#/ ab	a	a#/ bb	b	c

ex.: note 119 5 will play a middle c for half a second.

Using Sensors

- ➢ <u>The SUPER Cricket has 6 input or sensor ports, named a, b, c, d, e & f. You can use different types of input devices in these ports:</u>

- 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 primitives for reporting sensor values:

- *switch*, which reports a true-or-false reading – **when you are using a switch in a sensor port, you call it switcha, switchb…switchf.**

- *sensor*, which sends a value between 0 to 255 back to the Cricket – **when you use a senor, you call it sensora, sensorb…sensorf.**

switcha	Reports true if the switch plugged into sensor port a is pressed, & false if it is not pressed.
not switcha	Reports true if the switch plugged into sensor port a is released or NOT pressed, & false if it is pressed.
sensora	Reports the value of sensor a, as a number from 0 to 255*.
loop [send sensora]	This command will allow you to determine the range of sensor values that you need to write into your program, or to check your sensor readings. Type this line into the command center (for sensora or sensorb) & hit return to start sending sensor values back to your computer. **When you use the send command, the values will be displayed in the Cricket's monitor window.** Press the Cricket's run/stop button to stop the loop from running.

*** NOTE: when using a light sensor, the lower the reading the brighter the light** (just the opposite of what you would think). When the light sensor is flooded with bright light, it reads 0; when it is very dark, the reading will approach 255.

Control Statements

Control statements are commands used in programs to specify when and how things happen.

- **<u>REPEAT COMMAND</u>: executes the commands contained inside the bracket *x* number of times**

  repeat x [*commands to be executed***]**

  ```
  to whip.lash
  repeat 5 [a, thisway onfor 10 thatway onfor 10]
  repeat 5 [beep wait 5]
  end
  ```

- **<u>LOOP COMMAND</u>: executes the commands contained inside the bracket over and over, indefinitely**. In order to stop a loop from running, you must press the Cricket's run/stop button.

  loop [*commands to be executed***]**

  ```
  to glowlight
  loop [a, setpower 2 onfor 5
          setpower 4 onfor 5
          setpower 6 onfor 5
          setpower 8 onfor 5
          setpower 6 onfor 5
          setpower 4 onfor 5
          setpower 2 onfor 5
          wait 5]
  end
  ```

- **<u>RECURSIVE PROCEDURES</u>: another way that you can create a procedure that keeps repeating itself indefinitely is by calling out the procedure's own name as the very last instruction before the end statement.**

  ```
  to alarm
  a onfor 2
  beep
  alarm
  end
  ```

- **<u>WAITUNTIL STATEMENT</u>: this command "freezes" the program from processing further commands until the condition in the brackets is met** – with this command the Cricket is continuously checking to see if the condition is true, and when it does become true, the program execution will continue to the next line or command. <u>Note that the condition must be contained in square brackets</u>.

waituntil [*condition*]

The following procedure waits for a pushbutton switch (a) to be pressed and then it turns on motor a; it then waits for the switch to be pressed again in order to turn the motor off. <u>Note that it is often necessary to use a time delay in a program like this, because the Cricket processes command lines very quickly. Without the wait command, if you press the switch too slowly, it may already be reading the second waituntil statement and it will shut the motor off without ever turning it on</u>.

```
to turn-on-off
waituntil [switcha]
a, on wait 5
waituntil [switcha]
off
end
```

ex: The following procedure turns on an alarm when a door opens and a pushbutton switch is released (not pressed).

```
to turn-off
waituntil [not switcha]
loop [a, onfor 2 beep wait 1]
end
```

ex: The following procedure turns on a lamp when it gets dark:

```
to nightlight
waituntil [sensora > 50]
b, setpower 8 on
waituntil [sensora < 50]
b, off
nightlight
end
```

➢ **<u>USING SUBPROCEDURES</u>: once you write a procedure, you are actually defining a new command that your Cricket will recognize and know how to do - and you can use that procedure as part of any other procedure just by calling out its name.** When writing longer, more complex programs, it is good programming technique to break down a complicated task into individual tasks, and define a subprocedure for each task. You can then use these subprocedures inside the large main procedure in order to make it easier to read and understand; all your procedures should have meaningful names. You can also use any procedure that you define inside several other programs.

Ex. The following procedure uses the alarm procedure that you wrote above, as a subprocedure. Suppose a priceless piece of artwork hanging on a wall in a museum was pushing against a pushbutton switch; when the thief goes to steal the painting, the switch is released.

```
to catch.thief
waituntil [not switcha]
alarm
end
```

- **<u>IF STATEMENT</u>: the program will check to see if the required *condition* is true, and if so, it will execute the commands contained in the bracket. If the condition is not true, the program will skip the commands in the brackets and will run the next line of the program**. (note: A conditonal expression that evaluates to zero is considered false; non-zero conditonal expressions are considered true).

  if *condition* [*commands to be executed***]**

  ex: The following procedure turns on a motor and then automatically turns the motor off if someone puts their hand too close to it, and blocks the light sensor. <u>Note: you often need to put if statements inside a loop or a recursive procedure, otherwise, the Cricket will only check once to see if the condition is true and then the program will end.</u>

```
to safe.machine
waituntil [switcha]
 a, on
 loop [
 if sensorb > 50 [a, off]
 ]
end
```

- **<u>IF-ELSE STATEMENT</u>: the program will check to see if the required *condition* is true, and if so, it will execute the commands contained in the FIRST bracket; otherwise, if the condition is not true, the program will execute the commands contained inside the SECOND bracket.**

  ifelse *condition* [*commands to run if TRUE***] [***commands to run if FALSE***]**

  ex: This procedure repeatedly tests the state of a switch. If it’s pressed, the motor will run, but when the switch is released, the motor will stop:

```
to drivemotor
ifelse switcha [b, on][b, off]
drivemotor
end
```

ex: This procedure repeatedly tests the state of a light sensor. If it is seeing light, a green LED will turn on, but when the light sensor is covered and seeing darkness, a red LED will turn on.

```
to light.control
loop [
ifelse sensora < 20 [a, on][b, on]
 ]
end
```

- **<u>USING THE "AND OPERATOR" WITH IF STATEMENTS</u>: you can use AND as part of an if or if-else statement to make your Cricket CHECK TO SEE IF TWO CONDITIONS are BOTH TRUE at the same time.** For example, imagine that you have a car with two light sensors for bumpers (plug motor in output port a; plug photocells in sensor ports a & b). If the car is driving forward and it approaches a wall, both light sensors will sense less light; when BOTH sensors "see dark" as the car is about to hit the wall, the car will back up.

```
to lightdrive.1
a, setpower 8 on
loop [
ifelse sensora > 50 and sensorb > 50 [thisway] [thatway]
]
end
```

- **<u>USING THE "OR OPERATOR" WITH IF STATEMENTS</u>: you can use OR as part of an if or if-else statement to make your Cricket CHECK TWO CONDITIONS TO SEE IF EITHER ONE IS TRUE.** Again imagine that you have a car that you want to control using light sensors on the front bumper. This program makes the car back up when EITHER light sensor detects an obstacle:

```
to lightdrive.2
a, setpower 8 thisway on
loop [
if sensora > 50 or sensorb > 50 [thatway]
]
end
```

- **WHEN STATEMENT:** You can get your Cricket to do simple multi-tasking, which means that it can be running two programs at the same time. In addition to the primary task that is being executed, the Cricket can have another program that is running in the background (behind the scenes). **The "background task", that is started up using the WHEN command, repeatedly checks to see if a condition is true. When the condition becomes true, the when statement interrupts the primary task (main program) to execute a "special action"; once this action is finished, the primary task will continue running, picking up where it left off. The special action associated with the when statement is executed only once each time the condition becomes true.**

  ex. Suppose there is a continuous carnival ride that goes all day – it only stops if someone presses a button to signal that they want to get on. You could use a when statement for this program:

```
to constant.ride
when [switcha] [beep a, off]
loop [a, onfor 10 rd]
end
```

  In this example, the primary task that is running is contained in the loop statement, and the when statement defines the background task that is running:

when [switcha] [beep a, off]

the condition to be checked **the "special action" to be executed**

  Note that both the condition and the special action must be contained in brackets. It is also important to note that **a when statement must always precede an infinite loop**, otherwise, the program would never get to the when statement! The when statement itself initiates the background task, and it should not be placed in a loop - it only needs to be executed once.

  There can be only one background task operating at any given time. If you tried to execute another when statement after a background task has already been started, the subsequent when statement will cancel the earlier one.

  To cancel a when statement and stop a background task from operating, use the command*: whenoff*

OTHER CONTROL COMMANDS:

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.

Using Local Variables

- Procedures can be defined to accept inputs, which then become local variables inside the procedure. **Local variables are defined right after the procedure name, using the colon character ":". More than one local variable can be used in the same procedure (leaving a space between each variable). In order to run a procedure with a local variable, you must type the procedure name along with a value for the variable.**

- Here is an example of a procedure that uses a local variable. This flash procedure requires you to enter an input value, which tells the number of times to execute the repeat loop. You would run this flash procedure by typing flash 5, flash 20, etc. - the number entered for "n" will be the number of times the motor is turned on.

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

- Here is an example of a procedure that uses two local variables to control the speed of a motor and the duration of time it is on:

```
to ultimate.motor :speed :time
a, setpower :speed onfor :time
end
```

Using Global Variables

- **Global variables are variables that are always recognized by the cricket and can be used in any procedure. Global variables are created at the beginning of the procedures buffer using the command:**

global [*list of variable names*]

ex. global [rotations presses]

For example, this statement creates two global variables, named rotations and presses. **Once a global variable such as rotations is defined, another global-setting primitive, setrotations can now be used to assign values to the variable.** Thus, after these global variables are defined, we could write the following lines as part of any procedure:

setrotations 3	to set the value of rotations to 3
setpresses presses + 1	to increment the value of presses by 1 (takes the previous value of the variable presses and adds 1)

<u>Note:</u> that you do not use a colon (:) in front of the name of a global variable – this is only done for local variables!

- **Procedures can be written to assign values to global variables, using the "output" primitive.** In the following example, the variable "temp" is defined, and the detect procedure returns a value of 0, 1, or 2 for "temp" 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
```

In this procedure, a reading of sensor a is loaded into the variable "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. **You must insure that if a procedure sometimes produces an output, that it always does. In a procedure that uses the output command, the Cricket will crash if a value is not produced.**

<u>Using Timers and Counters</u>

- There are several timing primitives that are built into the Cricket; they are useful to cause the Cricket to do something for a length of time.

wait 20	<u>Delays program execution for 2.0 seconds, where time is given in tenths-of-seconds</u>. For example, wait 15 causes a delay of 1.5 seconds before the program will continue and run the next command.

- The Cricket also has a built in **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:

timer	<u>Reports value of timer in milliseconds = thousandths of seconds. So if timer reports 1000, that would indicate 1.000 second of elapsed time; if timer shows 2350, that would mean 2.350 sec</u>. Note that this is different from the other time commands – onfor and wait – which specify time in tenths-of-seconds. <u>Also note that the timer is only updated every 4 milliseconds</u>, so if you were watching it continuously, you would see it read out 0, 4, 8, 12, ... millisecs.
resett	Starts the timer and resets the elapsed time to zero.

- **<u>Making an Event Timer</u>**: Here is a program that can time a race. You need to set up 2 light beams: one for the starting line and one for the finish (plug two 2 lamps into output ports & 2 photocells into sensor ports. When the runner breaks the starting line the timer starts, and when they cross the finish line, the time is displayed in the cricket monitor window.

```
to time.race
ab, setpower 8 on
waituntil [sensora > 10]
resettimer
beep
waituntil [sensorb > 10]
send timer
beep
end
```

- **<u>Built-In Counters</u>: input/ sensor ports C through F have built in counters (ports a & b do not).** These ports can automatically keep track of the number of changes in the state of the sensor (i.e. number of times a switch is pressed or number of times a light sensor is covered). To use these built-in counters, you use the following commands:

resetcounterc (d, e, f)	starts the counter & resets it to 0
send counterc (d, e, f)	displays the counter value in the monitor window
if counterc = 10 [a, on]	you can use the counter value in control statements

- <u>**Making a Counter**</u>: In some cases, you may need to keep track of how many times a sensor changes state in ports a or b, which do not have built in counters. Here is how you write a counter program. Note that this program **requires you to define a global variable named *counts*** before you actually write the procedure. Every time the switch is pressed, the counting program “increments the variable” count, which means that it adds 1 to the variable to get a new value.

```
global [count]

to count.presses
loop [
waituntil [switcha]
setcount count + 1
waituntil [not switcha]
if count = 10 [a, on]
 ]
end
```

<u>Numbers & Operations</u>

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

- <u>**The Cricket DOES NOT USE STANDARD ORDER OF OPERATIONS**</u>. Instead, <u>mathematical operations are evaluated in the order in which they are encountered from left to right</u>. Thus 3 + 4 * 5 evaluates to 35, because the compiler first acts on the 3 + 4 and then multiplies this sum by 5. You may use as many parentheses as you need to in order to insure that mathematical expressions are evaluated the way you intend them to be.

- <u>The following table lists the operators provided in Cricket Logo</u>:

+	addition
-	subtraction
*****	multiplication
/	division
%	modulus (remainder after integer division).
and	AND operation
or	OR operation
not	NOT operation (logical inversion)
random	Reports pseudo-random number from -32768 to +32768. Use the modulus operator to reduce the range: **ex. random % 100 yields an integer from 0 to 99; random % 3 - yields 0, 1 or 2**

Infrared Communication

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
```

NOTE: 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.

Controlling Servo Motors

- The Super Cricket can control 8 servo motors - **servo port numbers 1 to 8**.
- You only need 2 commands to control servo motors – one to set the position (which automatically turns it on), and one to turn the servo off:

servo 1 100	turns servo motor 1 on and sets it to the **specified *position*, which can be any number between 0 – 255** (which roughly corresponds to 90 degrees of movement or 180 degrees with some servo motors)
offservo 1	this command turns off electricity to servo 1

- When using a servo motor, you will always start with the position command to tell the motor to turn on and to send it to its starting position. You will conitnue to use the servo position command to make the motors move to the desired positions to accomplish your task. Servo motors are very strong and when they are commanded to assume a position, electricity will continue to be sent to the motors so that they fight to hold their position.
- When you are done using a servo in your program, use the servooff command to shut off the power to the motors.

Using the LED Display

- Plug the LED display into one of the bus ports on the Cricket - you can use either bus port because they're both the same. Turn on the Cricket, and the display should show a rectangle of illuminated segments. This is its power-on indicator.
- **Procedure to Display Numbers:** copy this into your procedures buffer and download it to your Cricket. Then **in order to display a number, you simply type the display command followed by the number** (ex. display 7).

```
to display :n
```

```
bsend $110
bsend 0
bsend high-byte :n
bsend low-byte :n
end
```

- **<u>Procedure to Display Sensor Values</u>**: if you plug a photocell into sensor port a, you can use this command in a program, or in the command center, to continuously show the value of the sensor:

  ```
  loop [display sensora wait 1]
  ```

  In this case, the sensor value displayed will be updated 10 times per second, and you can control the number of updates per second by changing the wait time.

- **<u>Controlling Display Brightness</u>:** When the display is first turned on, it uses a mid-range power level that is a good compromise between brightness and battery drain. However, you can adjust the brightness level up or down, ranging from 1 (least brightness) to 7 (highest brightness). When the display is first turned on, it is automatically set to brightness level 4. Copy the following procedure into your procedures buffer, and then you can control the brightness level in your command center (ex. brightness 6)

  ```
  to brightness :n
  bsend $110
  bsend $80
  bsend 0
  bsend :n
  end
  ```

- In order to preserve battery power, the display will automatically turn off one minute after the last time it receives a command. If you have an application that requires the display to be on continually, simply put the display command into a loop so that it's issued more than once a minute, and your display will stay on.