Getting Started With
Handy Boards and Handy Logo

by

Robbie Berg
Department of Physics
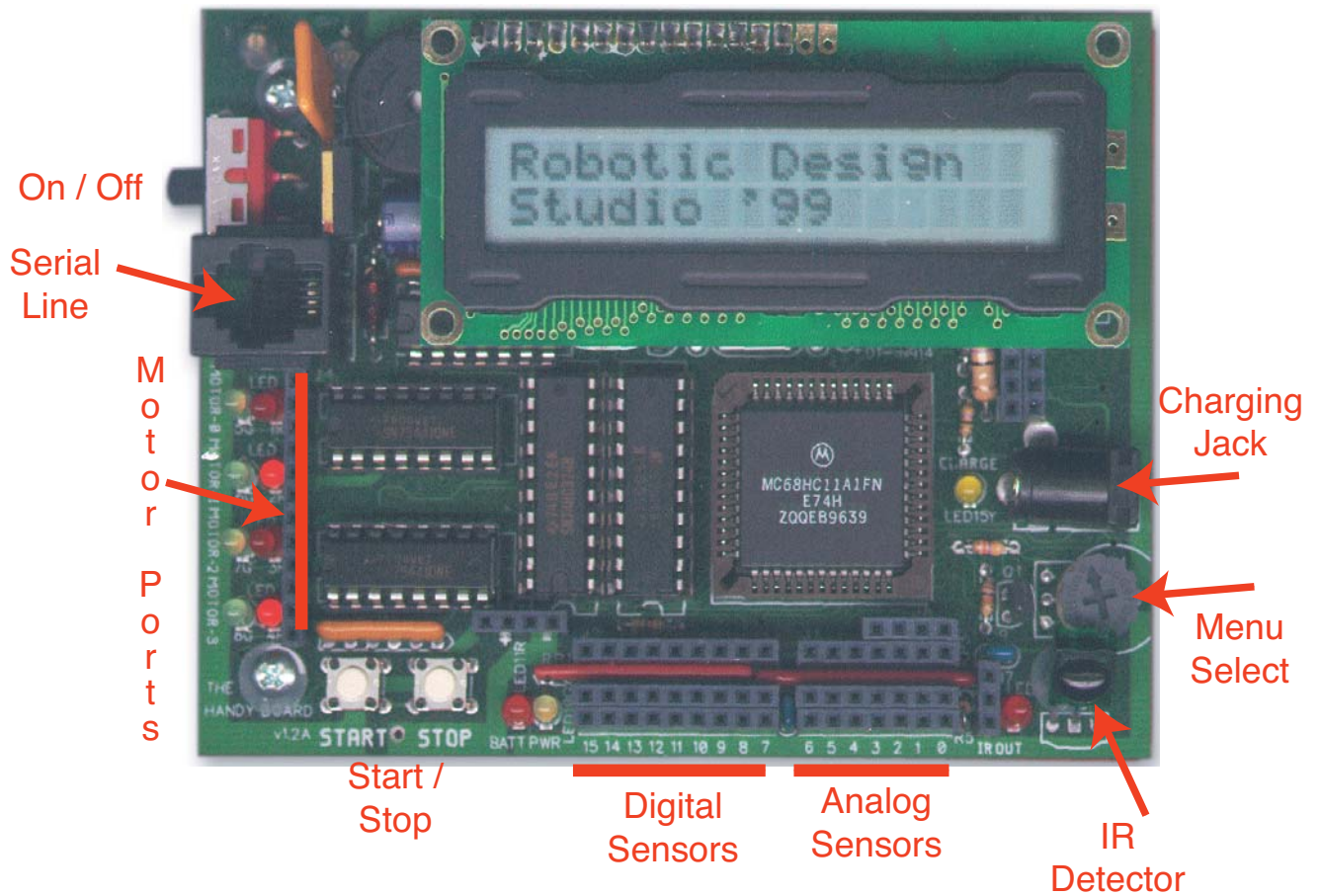
and

Franklyn Turbak
Department of Computer Science

Wellesley College

Handy Board Parts

On / Off

Serial
Line

Motor Ports

Start /
Stop

Digital
Sensors

Analog
Sensors

IR
Detector

Charging
Jack

Menu
Select

Robotic Design
Studio '99

MC68HC11A1FN
E74H
ZQQEB9639

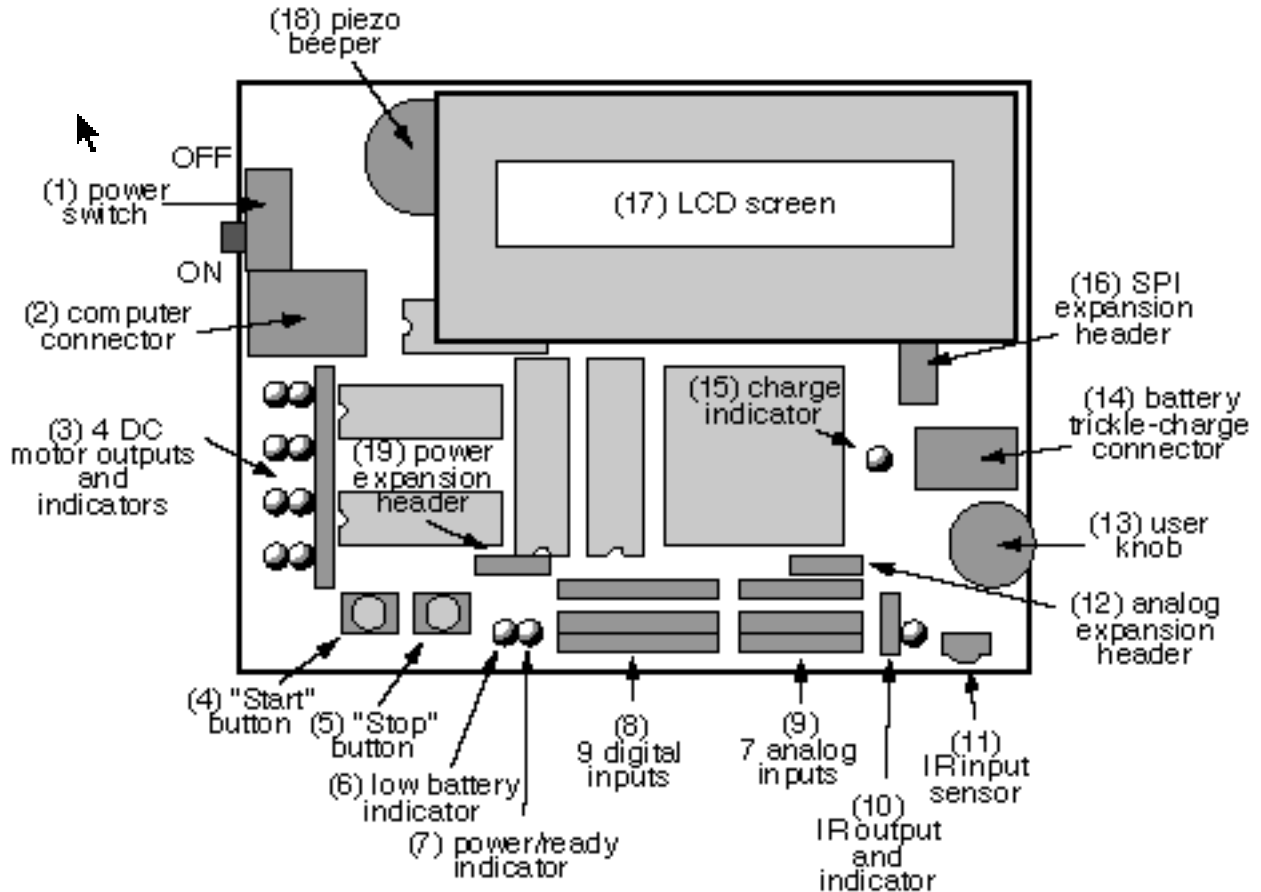# Getting Started With
Handy Boards and Handy Logo

Figure 1

1. Setting Up Your Handy Board

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 small motors. It is designed for a variety of educational robotics uses, including mobile robot projects, kinetic sculptures, data-taking applications, and "ubiquitous computing" applications (projects that embed computers in the world around us). A schematic drawing of the Handy Board is shown in figure 1 above.

To use your Handy Board you will need to transfer computer instructions from a computer to the Handy Board. The connections for downloading Handy Logo programs to the Handy Board are shown in figure 2 . Instructions are sent to the Handy Board via a cable that is connected to the computer's serial/modem port. This cable is connected to a small electronic device called the Serial Interface/Battery Charger (SIBC), and a telephone cable completes the connection between the SIBC and the Handy Board. ***Warning****: never use a telephone cable to connect two Handy Boards or to*

connect a Handy Board to a telephone outlet; serious damage can result from such connections! Power is provided to the SIBC (and indirectly to the Handy Board) via an AC adapter ("wall wart") plugged into a standard electrical outlet. It is also possible to power the Handy Board directly using the AC adapter.

In order to control the Handy Board, make sure it is turned on; you should hear a "boot beep" when you turn on the Handy Board and the green power light should come on. (If you don't see the green light, it probably means that the battery is dead. If you don't hear the beep, try turning off the Handy Board, waiting a couple seconds and turning it back on.) If you plan to use motors or sensors in your project, you should connect them to the motor ports (labeled `motor 0` through `motor 3` on the Handy Board) and sensor ports
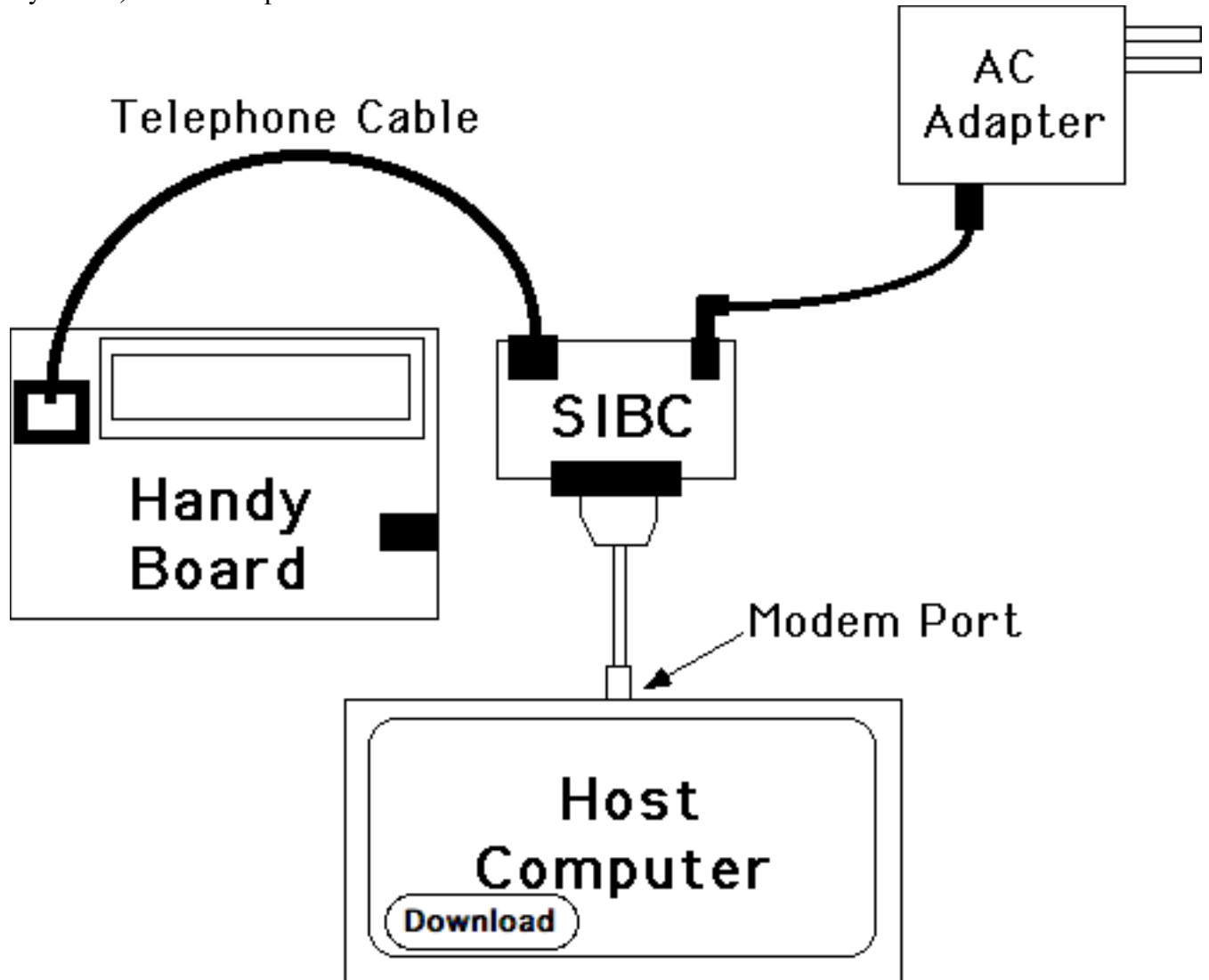

Figure 2 - Connections for downloading programs to the Handy Board

2. Programming Your Handy Board

You program your Handy Board using a special version of Logo called Handy Logo. The current version of Handy Logo is a Java application that you launch by double-clicking on the **HandyLogo.jar** file. The basic Handy Logo interface is shown in Figure 3.
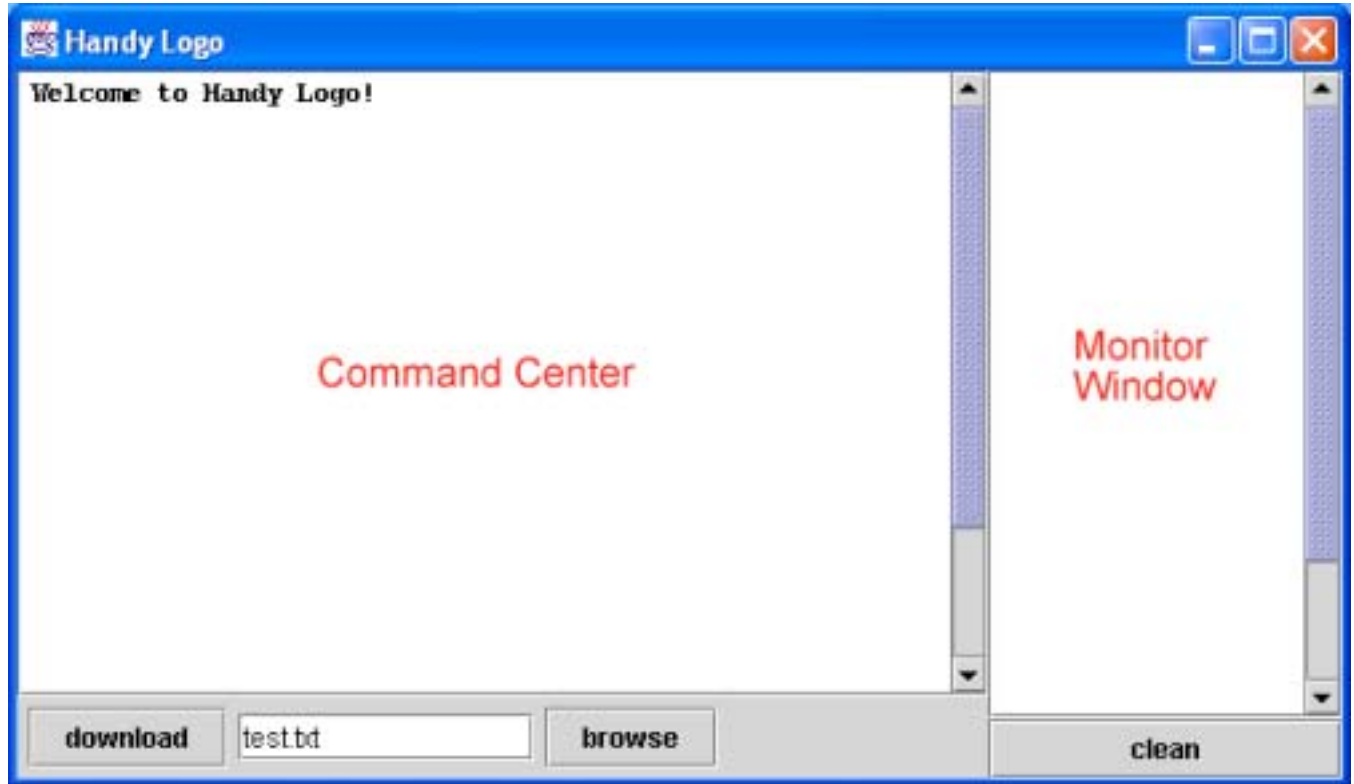

Figure 3 - The Handy Logo Window

To send an instruction to the Handy Board, simply type the instruction in the Command Center. For example, type beep in the Command Center and press Return. The instruction will be sent (via the serial cable) to the Handy Board, which will execute the command—and you should hear a beep.

It is not uncommon to receive a **"Handy Board not connected"** message when you attempt to send an instruction to the Handy Board. Often this message occurs because the connections between the computer and the Handy Board are not correct or because the Handy Board is not receiving adequate power (*e.g.* it is turned off, or the battery is low). If you get this message, make sure the Handy Board is turned on and the telephone cable is firmly connected to both the Handy Board and the SIBC. A sure sign of a good connection is that the yellow CHARGE light on the SIBC is on. Another source of this error is that two applications are competing to use the serial port. This frequently happens when you have launched more than one instance of Handy Logo. Make sure that there is only one Handy Logo window (or task-bar icon) before proceeding. Even if you have checked all of the above, you may still encounter some spurious "Handy Board not connected" messages. Often, simply turning the power off and on and resending the instruction will (eventually) result in a success. If not, try another Handy Board and/or SIBC.

You can type any Handy Logo instruction in the Command Center, and it will be immediately transferred to the Handy Board and executed. Below we give examples of some simple commands. A

full listing of Handy Logo commands is included in a separate handout called *The Handy Logo Language Reference*.

Plug motors into ports A and B on the Handy Board, then try these commands (pressing the Return key after each command):

`a, on`	Turns on the motor plugged into port A
`rd`	Reverses direction of the motor
`off`	Turns off the motor
`onfor 20`	Turns on the motor for 2 seconds
`repeat 4 [onfor 10 wait 10]`	Turns motor on and off 4 times
`ab, on`	Turns on motors in both ports A and B
`b, rd`	Reverses direction of motor in port B
`ab, off`	Turns off both motors

Now plug a touch sensor into digital switch port 7 and a light sensor into analog sensor port 0, and try these commands:

`waituntil [switch 7] onfor 20`	Turns on motor when touch sensor pressed
`on waituntil [switch 7] off`	Turns off motor when touch sensor pressed
`on waituntil [sensor 0 < 100] off`	Turns off motor when light sensor blocked

2.1 Writing Handy Logo Programs

The Command Center is convenient for executing short commands, but more complex robotic behavior is usually specified via programs consisting of *procedures*. A Logo procedure allows you to give a name to a sequence of commands and then execute the sequence by invoking the command. For example, here is the definition of a Logo procedure that gives the name **double-beep** to the three commands **beep wait 2 beep**:

```
to double-beep
    beep wait 2 beep
end
```

Procedures allow you to extend the "vocabulary" of commands that Logo understands with your own commands. A collection of procedures is called a *program*.

You can write a Handy Logo program by using any text editor to create a file that contains any number of procedure definitions. Examples of text editors are TextPad (PC), NotePad (PC), BBedit

(Mac), and Emacs (PC, Mac, Linux). Since Handy Logo programs must be simple text files it is important that you *not* use word processing programs such as Microsoft Word to create your programs.

In order to "teach" the Handy Board the procedures in your program, you need to *download* your program file to the Handy Board. To do this, first type the name of your file into the text box to the right of the **download** button below the Command Center. Alternatively, you can select the name of a file using the **browse** button. (Note that if you do not use the **browse** button to select a file, then the file to be downloaded must be in same directory as the **HandyLogo.jar** file.) Then click on the **download** button, which sends the file to the Handy Board. If your file contains the **double-beep** procedure, it will be downloaded to the Handy Board—but not executed yet. To execute your new procedure, type **double-beep** in the Handy Logo Command Center and press Return. Notice that the Handy Board's green light goes on while the Handy Board is executing the program. (Note that you can only download procedures when the green light is off—that is, when the Handy Board is not executing a program.)

2.2 Using the Handy Logo Menus

Turning the user knob on the Handy Board scrolls through a list of menu items. You can specify the menu items using menu declarations at the top of your program. For example, here are three menu declarations:

```
menu 1 [double-beep]
menu 2 [double-beep wait 2 double-beep]
menu 3 [a, on]
```

The square brackets associated with a menu item may contain any sequence of commands. When you download a program with these menu declarations, you can scroll through them with the user knob. You may declare up to seven menu items.

You can execute the commands in the currently displayed menu item by pressing the white START button on the Handy Board. For example, if you scroll to menu item 1 (**double-beep**) and press START, the double-beep procedure will be executed. An asterisk will appear on the screen to indicate that the procedure is running. Pressing the white STOP button will immediately stop *all* procedures from running.

Menu items are important because they allow you to execute programs even when the Handy Board is not connected to a desktop computer. This capability is essential for creating "autonomous" robots. ***You can now take the Handy Board anywhere!***

2.3 Using the Handy Board's LCD screen

You can information on the Handy Board's screen using the **print** instruction. For example try

print "hello	displays the word "hello" on the screen
print [hello world]	displays the phrase "hello world" on the screen
loop [print sensor 0 wait 1]	displays the value of the sensor plugged into analog sensor port 0, updated 10 times a second

Being able to view sensor data on the screen is incredibly useful in trying the figure out what is going on with your robot, so useful in fact that Handy Logo has a "built-in" ability to view the current

values of all seven analog sensor ports at once: If you dial the `user knob` all the way past menu item (7) you will see these sensor values displayed on the screen.

The **say** instruction allows you to display numbers and text in the Monitor Window on the right side of the Handy Logo window.

`say [hello world]`	displays the phrase "hello world" in the Monitor Window
`loop [say sensor 0 wait 1]`	displays in the Monitor Window the value of the sensor plugged into analog sensor port 0, updated 10 times a second

2.4 Some useful programming idioms

Procedures can accept arguments using Logo's colon syntax:

```
to arf :times
     ab,
     repeat :times [on wait 20 rd]
end
```

Procedures may return values using the output primitive:

```
to go
     ab,
     repeat third [on wait 10 rd]
end

to third
     if sensora < 20 [output 1]
     if sensora < 50 [output 2]
     output 3
end
```

The go procedure will execute 1, 2, or 3 times depending on the value of sensor A.

Data recording and playback

There is a single global array for storing data which holds 8K two-byte numbers. There is no error checking to prevent overrunning the data buffer. The essential primitives for data taking are:

record *value* - records *value* in the data buffer and advances the data record pointer.

recall *value* - reports the *value* of the current data point and advances the data playback pointer.

erase - Resets the value of the record pointer to zero.

resetr - Resets the value of the recall pointer to zero.

For example the procedure **take-data** can be used to store a *number* of data points recorded by sensor A once every second:

```
to take-data :number
    erase
    repeat :number [record sensora wait 10]
end
```

while the procedure **playback-data** can be used to send the recorded back to a host computer over the serial line

```
to playback-data :number
    resetr
    repeat :number [send recall sensora]
end
```

Multi-Tasking

Handy Logo has a number of different primitives for supporting multitasking. For example

forever [*action*]	launches a process to repeatedly execute action
when [*condition*] [*action*]	launches a process to repeatedly test *condition* and execute action when it is true
every [*time*] [*action*]	launches a process to execute *action* every *time* tenths of a second
stoprules	stops all running processes

For example, suppose a motor is connected to port A and a touch sensor to digital sensor port 7. Note the behaviors obtained with the following different procedures:

```
to wiggle-and-beep-when-bumped
    forever [a, onfor 2 rd]
    when [switch 7] [beep]
end


to wiggle-until-bumped
    forever [a, onfor 2 rd]
    waituntil [switch 7]
    stoprules      ; stops the "forever" rule from running
    beep
end


to wiggle-and-beep
    forever [a, onfor 2 rd]
    every 10 [beep]
end
```

Edge-triggered vs. level-triggered logic

Although the **waituntil** primitive is "level-triggered" the following example shows how to use **waituntil** to trigger an action on the edge of an event.. Assume a touch sensor is plugged into the digital sensor # 7 port.

```
to beep-once-per-press
    waituntil [not switch 7]
    waituntil [switch 7]
    beep
    beep-once-per-press
end
```

Alternatively, the **when** primitive is inherently edge-triggered, so another way to do this is simply:

```
to beep-once-per-press
    when [switcha] [beep]
end
```

3. Handy Logo Sampler

A Simple Program

Here's a simple program written this summer by two 10 year old kids who wanted to build a dancing robotic creature:

```
to dance
    cha-cha-cha
    go-round
    shake-it
end

to cha-cha-cha
    repeat 4 [back-and-forth]
    ab, off
end

to back-and-forth
    ab, thisway onfor 3
    beep
    ab, thatway onfor 3
    beep
end

to go-round
    a, on thisway
    b, on thatway
    beep wait 1 beep wait 1 beep
    wait 60
    ab, off
end

to shake-it
    a, thisway
    b, thatway
    ab,
    repeat 10
        [beep
        onfor 1
        beep
        rd
        onfor 1
        rd]
end
```

The Wandering LEGObug: An example with sensors

The LEGObug, is a creature with two motors connected to its two rear wheels. It also has two touch sensors connected to two "whiskers" positioned on either sides of its head and two light sensors that serve as "eyes".   Detailed plans for building the LEGObug are available at the following URL:

`http://lcs.www.media.mit.edu/people/fredm/projects/legobug/`

The procedure **seek** shown below causes the creature to be attracted to bright light. It assumes that the light sensors are plugged into the Handy Board's sensor-ports "0" and "1". The light sensors have the property that the greater the amount of light that hits them, the smaller the sensor value that is produced. (In typical indoor lighting the light sensors might give readings in the 15 - 30 range, if you shine a flashlight on them, they will produce a reading in the 1 - 5 range. It takes almost complete darkness to produce a reading of 255.) The program below causes the creature to move forward if bright light hits the light sensor plugged into sensor-port "0"

```
to seek
    loop [
        ifelse (sensor 0) < 10
            ;N.B. the parentheses are essential  here!!
            [go-forward]
            [stop-motors]
        ]
end

to go-forward
ab, on thisway
    ;the motors are each hooked up so that the
    ;"thisway" direction causes them to drive forward
end

to stop-motors
    ab, off
end
```

As an exercise you might try making creatures that run away from the dark, or ones that turn toward a bright light.

The procedure **wander** shown below causes the LEGObug to drive straight until a whisker bumps into an obstacle. (It assumes that the touch sensors are plugged into the two of the digital sensor-ports. (the left touch sensor is plugged into digital sensor-port "7" and the right touch sensor is plugged into digital sensor-port "8") In an attempt to avoid the obstacle, it the creature backs up a bit, turns a small (random) amount and continues to drive forward.

```
to wander
     go-forward
     waituntil [or (touch-left?) (touch-right?)]
     ifelse touch-left?
          [back-up turn-right]
          [back-up turn-left]
     wander     ;note tail recursive alternative to loop
end

to go-forward
     ab, on thisway
end

to touch-left? ;touch-left reports "true" if the sensor
          ;plugged into digital sensor-port "7" is pressed
     output switch 7
end

to touch-right?     ;touch-right reports "true" if the ;sensor
plugged into digital sensor-port "8" is pressed
     output switch 8
end

to turn-right
     ;turns right for a random amount of time between 0 and
     ;5 seconds.
     b, off 5
     a, thisway onfor (random 50)
end

to turn-left
     a, off
     b, thisway onfor (random 50)
end

to back-up
     ab, thatway onfor 20
end
```