



[www.gogoboard.org](http://www.gogoboard.org)

By Arnan (Roger) Sipitakiat  
Future of Learning Group, MIT Media Lab

## GoGo Board 3.0

The GoGo Board is a general purpose sensing and control device. You can use it as a brain for projects in Robotics, Game Controllers, Data Logging, Interactive Art, and much more. It can be controlled directly from a Computer via the serial cable (tethered mode). It can also run autonomously using the Cricket Logo language.

This version is basically the same as the old 2.3 version but the sensor port pin assignments have been changed. This change will ease the soldering process needed to make many sensors. See details in the what's new page. The 3.0 version number signifies this incompatibility with old sensors.

The GoGo board features a one-sided PCB for easy fabrication. The bill-of-materials are available. In the USA, the parts can be conveniently purchased from digikey.com. They should be available in many other countries as well. Assembling the board requires nothing more than basic soldering tools and the willingness to try.

## Getting Started With The GoGo Board

### What you need

The following is a list of things you need in order to use the GoGo board



A GoGo board



A 9V power adapter or a battery pack.  
See powering the GoGo board below  
for more information



Serial cable



Sensors and output devices (like  
motors, lights)

## Getting to know the GoGo board

There are a few parts in the GoGo board with which you need to be familiar. Figure 1 illustrates the basic GoGo board layout.

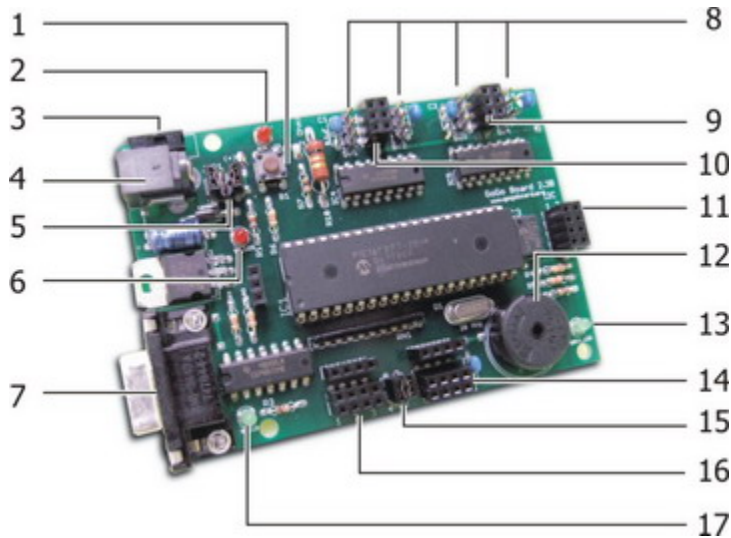


Figure 1: GoGo board layout

1	Start/Stop Button	Starts or stop the board's stored procedures.
2	Run LED	When lit, indicates that a stored procedure is running.
3	On/Off Switch	This is the main power switch.
4	Power Input	This is the main power input for the board. Typical input is 6-9 Volts. You can also power the board with the optional battery pack. See Powering the GoGo board below for more information.
5	Power Configuration	These jumpers allows you to configure how the motor output ports are powered.

	Jumpers	<p>Jumper 1 and 3 selected (default): Output is 5 volts</p> <p>Jumper 1 and 2 selected: Output voltage from the power jack is used</p> <p>See the power configuration document for more options.</p>
6	Power LED	This LED always lights up when there is power and the board is on.
7	Serial Port	The GoGo board uses this port to connect to the computer.
8	Servo Ports	Allows you to connect up to four servo motors.
9,10	Output A/B and Output B/C	Output Ports. You can connect to actuators such as motors, light bulbs, and relays to them.  Note that these ports share resources with the servo ports. Thus, ports that are used for servo motor control cannot be used as an output port at the same time.
11	Extension Bus	This port allows the GoGo board to communicate with other devices (such as an LCD display, a stepper motor controller)
12	Beeper	A programmable beeper.
13	User LED	A user programmable LED.
14,16	Sensor Ports	Allows connection to eight sensors (i.e. light, touch, temperature, distance, etc).
15	Sensor Configuration Jumpers	These two jumpers allows you to configure sensor ports 4 and 5. If jumper is removed, the port will bypass the voltage divider on the board. This allows the board to work with sensors that generate voltage (such as the <a href="#">Sharp proximity sensor</a> ).
17	Serial Activity LED	Indicates communication between the board and the computer.



## Testing the GoGo board

The easiest way to test the GoGo board is to use the GoGo board monitor program. You simply start the program and click on the "connect" button. If there are no error messages, it means you are connected! You can then try to read sensor values and turn on or off output ports. Figure 3 shows a screenshot of the GoGo board monitor software.

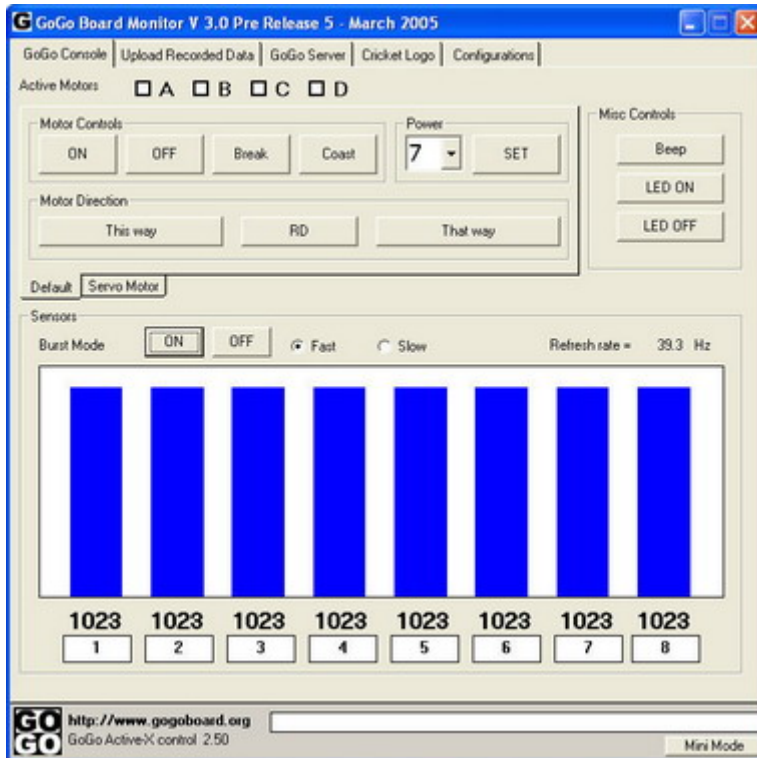
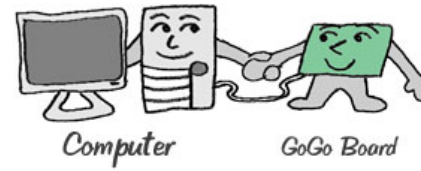


Figure 3: A screenshot of the GoGo monitor software

## Where to go next?

Now that the GoGo board is working, you can start using it in many programming environments. The GoGo board can function in two modes: tethered and autonomous.

### Tethered Mode



In this mode the board is always connected to the computer (via a serial cable). It allows any programming language that can access a serial port to interact directly with the sensor values and to actuate various devices. You can create games, interactive art work, and many other applications in this mode. Currently the GoGo board has libraries to support Microworlds Logo, Imagine Logo, Squeak, Active-X compatible languages (i.e. Visual Basic, Visual C++, MS-Office). Please visit the download page for more information.

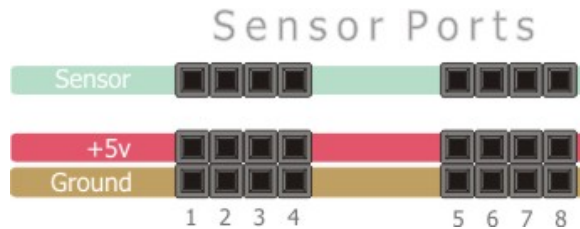
### Autonomous Mode



This mode allows users to download procedures to the board. Then, the board can execute those procedures while disconnected from the computer. This allows users to create autonomous robots, environmental sensing devices, and other smart objects. The language that is used to program the GoGo board is called Cricket Logo. There are many programs that supports this language. See GoGo board Cricket Logo support page for more information.

## Sensor ports on the GoGo board

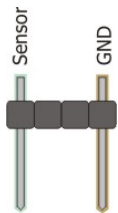
The GoGo board has eight sensor ports. Each of them has three pins, as shown in the illustration below. The top and bottom rows (Sensor Input and Ground) are the ones most simple sensors use. The middle row is an extra power supply for active sensors that needs a power source to function properly.



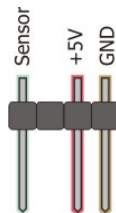
Note: this layout is different for older GoGo boards. If you are using GoGo Board version 1.x, 2.x (except 2.3B) then you must swap the "sensor" and "+5v" pins. Click [here](#) to see the old pin assignment.

## Active and Passive Sensors

Sensors that have been mentioned so far are called passive sensors. They do not need separate power to operate. Active sensors, on the other hand, are sensors that need their own power. An easy way to distinguish active sensors from passive ones is to count the number of pins it has. Active sensors have an extra third pin to get the power it needs while passive sensors have only two.



Connector for passive sensors



Connector for active sensors

Active sensors are more complex, but they open up a broad range of sensing possibilities. Examples of active sensors include Infrared sensors [it detects presence, distance], Hall effect sensors [detects magnetic field], noise sensors, vibration sensors, etc.

# Logo Reference (GoGo board edition)

## Motors

<b>a,</b>	selects motor A to be controlled.
<b>b,</b>	selects motor B to be controlled.
<b>ab,</b>	selects motors A and B to be controlled.
<b>c,</b>	selects motor C to be controlled.
<b>d,</b>	selects motor D to be controlled.
<b>cd,</b>	selects motors C and D to be controlled.
<b>abcd,</b>	selects all motors to be controlled.
<b>on</b>	turns the selected motors on.
<b>off</b>	turns the selected motors off.
<b>brake</b>	actively applies a brake to the selected motors.
<b>onfor</b> duration	turns the selected motors on for a duration of time, where duration is given in tenths-of- seconds. E.g., <b>onfor 10</b> turns the selected motors on for one second.
<b>thisway</b>	sets the selected motors to go the "thisway" direction, which is defined as the way that makes the indicator LEDs light up green.
<b>thatway</b>	sets the selected motors to go the "thatway" direction, which is defined as the way that makes the indicator LEDs light up red.
<b>rd</b>	reverses the direction of the selected motors. Whichever way they were going, they will go the opposite way.
<b>setpower</b> level	sets the selected motor(s) power level. Input is in the range of 0 (coasting with no power) to 8 (full power).

## Servo Motors

<b>setsvh</b> direction	sets the servo heading. direction for a typical servo motor should range between 15-45.
<b>svr</b> steps	turns right (counter clock wise) for a number of steps.
<b>svl</b> steps	turns left (clock wise) for a number of steps.

## Timing

<b>timer</b>	reports value of free-running elapsed time device. Time units are reported in 1 millisecond counts.
<b>resett</b>	resets elapsed time counter to zero.
<b>wait</b> duration	delays for a duration of time, where duration is given in tenths-of-seconds. E.g., <b>wait 10</b> inserts a delay of one second.

## Sound

<b>beep</b>	plays a short beep.
-------------	---------------------

## Sensors and Data Collecting

<b>sensor1</b>	reports the value of sensor 1, as a number from 0 to 1023.
<b>sensor2</b>	reports the value of sensor 2, as a number from 0 to 1023.
<b>sensor3</b> <b>sensor4</b> <b>sensor5</b> <b>sensor6</b> <b>sensor7</b> <b>sensor8</b>	same as sensor1 and sensor2, but report values of the corresponding sensor.
<b>switch1</b>	reports "true" if the switch plugged into sensor A is pressed, and "false" if not.
<b>switch2</b>	reports "true" if the switch plugged into sensor B is pressed, and "false" if not.

**switch3** same as switch1 and switch2, but report values  
**switch4** of the corresponding sensor.  
**switch5**  
**switch6**  
**switch7**  
**switch8**  
**resetdp** reset the value of data pointer to 0.  
**record** value records value in the data buffer and advances  
the data pointer.  
**recall** value reports the value of the current data point and  
advances the data pointer.  
**erase** sets the value of the first number elements of  
number the data array to zero and then sets the data  
pointer to zero. Because the process of  
recording data is relatively slow (about 20  
milliseconds per data point) it could take as long  
as 50 seconds for the **erase 2500** command to  
be executed.

## Control

**forever**[body] repetitively executes body indefinitely.  
**repeat** times executes body for times repetitions. times  
[body] may be a constant or a calculated value.  
**if** condition [body] if condition is true, the cricket executes  
body. Note: a condition expression that  
evaluates to zero is considered "false"; all  
non-zero expressions are "true".  
**ifelse** condition if condition is true, executes body-1;  
[body1] [body2] otherwise, executes body-2.  
**waituntil** loops repeatedly testing condition,  
[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.  
**output** value terminates execution of procedure,

reporting value as result.

## Numbers

**+** infix addition  
**-** infix subtraction  
**\*** infix multiplication  
**/** infix division  
**%** infix modulus (remainder after integer division)  
**and** infix logical "and" operation (bitwise and)  
**or** infix logical "or" operation (bitwise or)  
**xor** infix logical "xor" operation (bitwise xor)  
**not** prefix logical not operation. use only with boolean  
values (1 and 0).

**random** reports pseudo-random number from 0 to 32767.

## Variables

**make** "[variable] [value] Create variable  
**repeat** : [variable] Call variable

---

## Examples

**forever** Beep if sensor1's value is less than 500:

```
to example1
  forever
  [
    if sensor1 < 500 [beep]
  ]
end
```

**repeat** Beep 5 times with an interval of 1 second:

```
to example2
  repeat 5 [beep wait 10]
end
```

**if** Motor in port 'a' will turn on if sensor8's value is less than 1000; otherwise it will turn it off:

```
to example3
  forever
  [
    if sensor8 < 1000 [a, on]
    if sensor8 > 1000 [a, off]
  ]
end
```

**ifelse** Turns motor 'a' on, if sensor1's value is less than 500. Otherwise, the motor is turned off.

```
to example4
  forever
  [
    ifelse sensor1 < 500 [a, on][a, off]
  ]
end
```

**record data** Record sensor1's value every 1 minute for 1 hour

```
to example5
  resetdp
  repeat 60 [record sensor1
    beep
    wait 600
  ]
end
```

**display** Note that some characters can't be showed on a 7-segment display (i.e. X, W, Z). They will appear blank.

It will show the number '100':

```
to example6
  show 100
end
```

It will show the value of sensor1:

```
to example7
  show sensor1
end
```

It will show the characters 'hiho':

```
to example8
  show "hiho"
end
```

It will create the variable 'i' with a value of 100. Then the display will display the value of 'i,' equal to 100:

```
to example9
  make "i 100
  show :i
end
```

When the sensor8's value is less than 1000, it displays the word "low;" otherwise, it will display the value of sensor8:

```
to example10
  forever
  [
    if sensor8 < 1000 [a, on show "low]
    if sensor8 > 1000 [a, off show sensor8]
  ]
end
```

**variables** Creates variable 'x' with a value of '5.' Then it will repeat 'x' (5) times beep and wait 1:

```
to example11
  make "x 5
  repeat :x [ beep wait 1]
end
```

Will count the number of times sensor1's value becomes less than 500. It will beep for a number of times equal to the count value

```
to example12
  make "count 0
  forever
  [
    if sensor1 < 500
    [
      make "count :count + 1
      repeat :count [beep wait 5]
      waituntil [sensor1 > 500 ]
    ]
  ]
end
```

**setpower** Set power of ports 'a' and 'b' to 7. ab are on for 5 seconds and wait 2 seconds. Then set power to 2. ab are on for 5 seconds.

```
to example13
  ab, setpower 7
  wait 1

  ab, onfor 50
  wait 20

  ab, setpower 2
  wait 1
  ab, onfor 50
```

### nesting

Imagine a car. If both sensors are less than 300, the car will go forward. If both are less than 300 then it will go backwards. If one is less than and the other greater than, then it will turn right or left respectively:

```
to example14
  forever
  [
    ifelse (sensor8 < 300) and
      (sensor1 < 300)
    [straight]
    [
      ifelse (sensor8 > 300) and
        (sensor1 < 300)
      [left]
      [
        ifelse (sensor8 < 300) and
          (sensor1 > 300)
        [right]
        [
          if (sensor8 > 300) and
            (sensor1 > 300)
          [reverse]
        ]
      ]
    ]
  ]
end
```

```

to right
  b, off
  a, thisway on
end

to left
  a, off
  b, thisway on
end

to straight
  ab, thisway on
end

to reverse
  ab, thatway on
end

```

## machine state

When the system has only two states, so to differentiate between one state and the other, then a variable 'state' is created with value '0.' When it performs something then it changes the state to '1':

```

to example15
  make "state 0

  forever
  [
    waituntil [sensor1 > 70]
    wait 5
    if sensor1 > 70
      [
        ifelse :state = 0
          [ a, on make "state 1 ]
          [ a, off make "state 0 ]
      ]
    ]
  ]
end

```

## servo

Setsvh is "set servo heading." the number you give to it is not the "degrees" to which the servo will point. Which direction the motor will point will be different from brand to brand. For the Futaba one, 0 degrees is a bout 20 and 180 degrees is 40.

svl and svr are "servo left turn" and "servo right turn" accordingly. Again, the number you give to it is not the "degrees." For example, my motor "svl 1" will turn the motor  $180 / (40-20) = 9$  degrees.

If you servo has been modified to work in a "continuous" mode then these commands will determine how fast the motor will turn. I think this may be what you are facing.

```

to test
  forever
  [
    ifelse sensor1 < 500
      [a, setsvh 20]
      [a, setsvh 40]
    wait 1
  ]
end

```