RoboBricks Tutoral

Table of Contents

RoboBrick® Tutorial

Table of Contents:

Introduction

Why RoboBricks?

One of the challenges confronting the robotics hobbyist is that programming complexity does not rise linearly with each newly added sensor or behavior. Programming a robot with bump sensors and IR detectors to behave as a line follower is relatively simple. But add several more sensors or try to do something more interesting than just following a line and the programming challenges quickly become more than the average hobbyist can handle. This rapid rise in programming complexity associated with more advanced autonomous robots may be the single largest stumbling block to the advancement of this hobby.

RoboBrick technology offers one solution to this problem. By putting the "housekeeping" programs for each sensor or task in a dedicated RoboBrick, the top level "behavior" program becomes considerably easier to write. Those with very little programming experience can now concentrate on adding more interesting levels of behavior to their robots without being limited by their lack of programming experience. RoboBrick technology promises to expand the boundaries that presently limit the robotics hobbyist from developing robots with more complex and interesting behaviors.

About This Manual

This manual is designed for novice RoboBrick users to help them implement their robot inspirations using RoboBrick technology. The manual focuses on a "hands−on" approach to the implementation and programming of certain Brick modules specifically selected because they are likely be found in most robot designs. The technical information presented here is introductory in nature and therefore does not go into the level detail that is available at the Official RoboBrick Website. Once the user becomes more familiar with RoboBrick technology, the Official RoboBrick Website should become his/her resource of choice.

RoboBrick Philosophy

Make programming easier

Think of RoboBricks as an implementation of distributed programming. Each RoboBrick module contains a PIC microprocessor pre−programmed with the serial communications protocol, Brick identification data and all of the code necessary to perform each of the functions for which the Brick is designed. Consequently, the user is relieved of the burden of writing this difficult code for himself and can now turn his attention to developing a top level code that is significantly less complicated to write.

Reduce the need for circuit building

A side benefit derived from using RoboBricks is that the user is relieved of the chore of building the circuitry necessary to accomplish the particular task at hand. This reduces the chance of design and construction errors and eliminates the frustrations and delays associated with troubleshooting. RoboBricks therefore can be viewed as an approach to "plug and play" technology. The circuit diagrams for each of the Brick modules are available on the Official RoboBrick Website.

Be Physically compatible with Legos

The typical RoboBrick module is built on a rectangular circuit board having a width of 1.25 inches and a length of 2.50 inches. The widths and lengths of non−standard RoboBricks are always fabricated in multiples of 1.25 inches. Along its width, each RoboBrick board contains holes sized and spaced to interlock with Lego's standard plastic brick pegs thus allowing Lego creations to make use of the powerful technology available in RoboBricks.

Provide open source code

The source code for each RoboBrick, is available at the Official RoboBrick Website. The source code for each Brick is written in µCL (pronounced like "uncle without the "n"), a language developed specifically for RoboBricks and other robotics applications. More information is available at The µCL Project.

Be Microprocessor independent

While three different RoboBrick Hubs, each using a different microprocessor, are included in the RoboBrick set, most any other microprocessor will work equally as well. All that is required is that the user follow the simple electrical specifications and communications protocol.

Technical Information

Inter−Brick communication

A RoboBrick system consists of a single "Hub or Master" brick connected to one or more "Slave" bricks via individual 4−line cables. The cables provide +5 volt regulated DC power and ground from the Hub to each Slave along with a two separate lines for sending and receiving data. The number of Slave bricks that can be connected is limited by the Hub brick chosen. For example, the BS2Hub8 can control up to eight Slave bricks while the OOPicHub15 can control up to 15 Slave bricks (the number of slave bricks that any chosen microprocessor can support is approximately equal to the total number of in/out ports available divided by two). Asynchronous data transfer is done on a two line system where the lines are connected to specific input/output ports on the Hub and Slave microprocessors. Data is sent on one line and received on another; the Hub transmit line is also the Slave Receive line and the Hub receive line is the Slave transmit line. To address a selected Slave Brick one only needs to know the two microprocessor ports ("sockets" in RoboBrick speak) where the Slave is connected to the Hub's microprocessor. Communication between the Master and a Slave is always initiated by the Master; the Slave sends data to the Master only when the Master instructs. For example, if a Servo4 Slave is in use, the Hub might send a new speed to the Servo4 but not ask for any report back. The Servo4 Slave would execute the new command without a response. On the other hand, the Hub might request that the Servo4 Brick report its current speed argument in which case, the Brick will respond with the appropriate information. Communication is asynchronous; it proceeds first from the Master to the Slave and then from the Slave back to the Master using asynchronous, 8N1 protocol (i.e. 1 start bit, 8 data bits, no parity, and 1 stop bit.) at a 2400 baud rate.

Programming tables

Each Slave RoboBrick is designed with a specific function in mind. Every Slave Brick offers the user a broad variety of choices about how to employ its various functions and how to extract information about its status. To assist the user in taking full advantage of all of the functions available on a particular Slave brick, each Brick has its own programming table that lists each of the functions available and the respective command

byte that must be sent to the Slave to achieve it. For example, the LED10 Slave brick's programming table shows that this Brick has 13 commands available, 9 that control how it will function and 4 that will provide information about its status. Each of these 13 functions has a command with a unique byte value associated with it. In some cases, function commands to the slave brick will need a response from the Slave back to the Hub and in other cases it will not. When a response is required as part of a function command, the Slave will respond automatically without any additional action on the part of the user.

Interrrupt protocols

At 2400 baud, it can take a while to poll several input RoboBricks to see if anything interesting has occurred. Sometimes RoboBricks are sensing inputs that need a response that is faster than strict polling can provide. For example, bumper detectors. To support low latency, many RoboBricks support the RoboBrick Interrupt Protocol.

The RoboBrick Interrupt Protocol is very simple. Each RoboBrick that supports the protocol has two bits −− the interrupt pending bit and the interrupt enable bit. The interrupt pending bit is set by the RoboBrick when a pre−specified user event has occurred. The interrupt enable bit is set to allow the interrupt to occur.

In this example, we will not be using the Interrupt feature but you can read more about it on the Official RoboBrick Website under Interrupts.

Types of commands

There are three basic commands available for each RoboBrick Slave Module: Shared Commands, Information Commands and Function Commands. Shared Commands are commands common to every RoboBrick. These commands are used to get administrative information about the Brick such as its serial number, model and version number, etc. Information Commands are used to obtain the current operating status of a Slave Brick and might include, as an example, the status of each sensor connected to the Brick or the commands the Brick is sending to a connected device. Function Commands are those sent to the Slave to cause it to perform some action such as, for example, resetting the speed of a servo, changing the blink rate of an LED, turning a connected device on or off, etc.

Building a RoboBrick System

In this section, we will use RoboBricks to accomodate the construction of a mobile, autonomous, tabletop robot that possesses the following four behaviors:

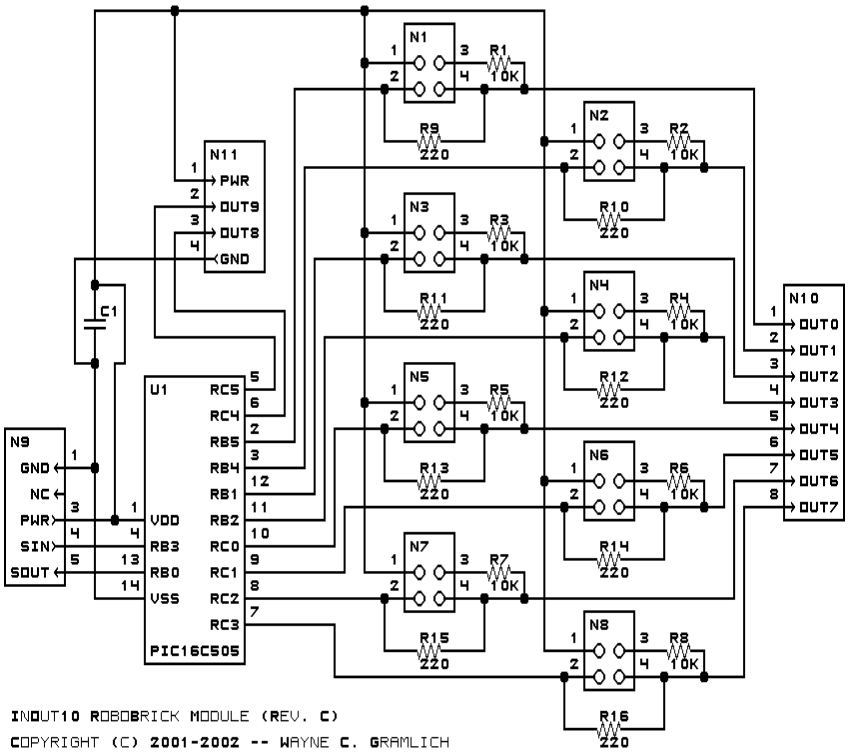
1. Able to randomly wander about its tabletop environment
2. Capable of avoiding objects in its path
3. Capable of escaping from unexpected collisions with an undetected objects
4. Capable of detecting and remaining clear of the table's edge

Since the purpose of this exercise is to learn how to use RoboBricks, we will focus on building and programming the RoboBrick system and ignore both the assembly of the robot platform and the details of how the Bricks attach to it. Remember that RoboBricks are designed to fit onto most any robot platform and they can be affixed in whatever manner the user decides. We will use servos, modified for continuous rotation, for locomotion; microswitches as bump sensors to detect collisions; a Devantech sonar to detect objects to be avoided; and four downward looking IR detectors to detect the table's edge. With that in mind, we have selected the following RoboBricks for our system: a BS2Hub8 to control the following four Slave bricks: Servo4, InOut10, SonarDT1, AnalogIn4 and Light4 . We will start by writing a small stand alone program to

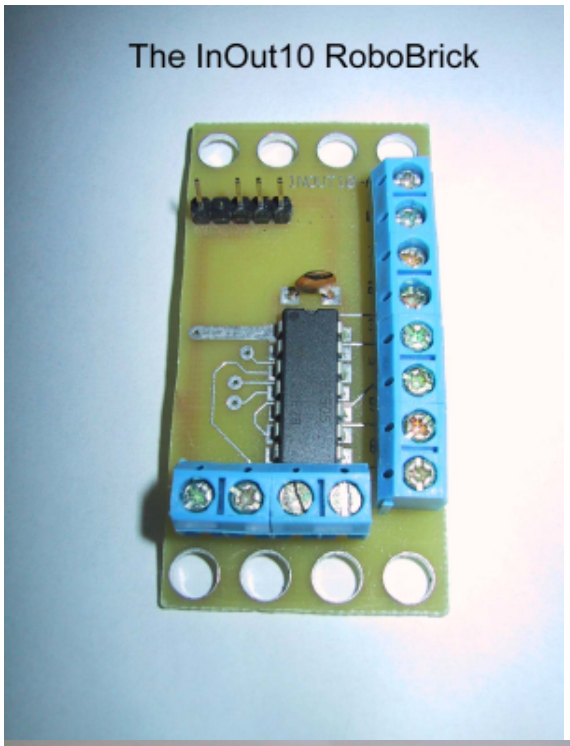
test each of the RoboBricks chosen together with whatever apparatus is attached. As a final step, we will use the essential parts of each of the test programs to write a Table Top Robot program. All of these programs can be downloaded at XXXX

The Bump Sensors

We will use two microswitches for our bump sensors and attach them to the InOut10 brick. This RoboBrick provides a +5 volt regulated power and ground connection and the ability to input or output up to 10 bits of data. It is perfectly suited to implement our bump sensors. We can configure each microswitch so that it responds with an active high or active low output whenever it is closed on the event of a collision. Arbitrarily, we will elect to configure the switch to respond with an active high output upon a collision. We'll use Terminal N3 (see below) to connect our bump sensors with the following pin assignments: pin 1 = +5V; pin 2 = left bump sensor; pin 3 = right bump sensor; pin 4 = ground.



Bump Sensor Construction



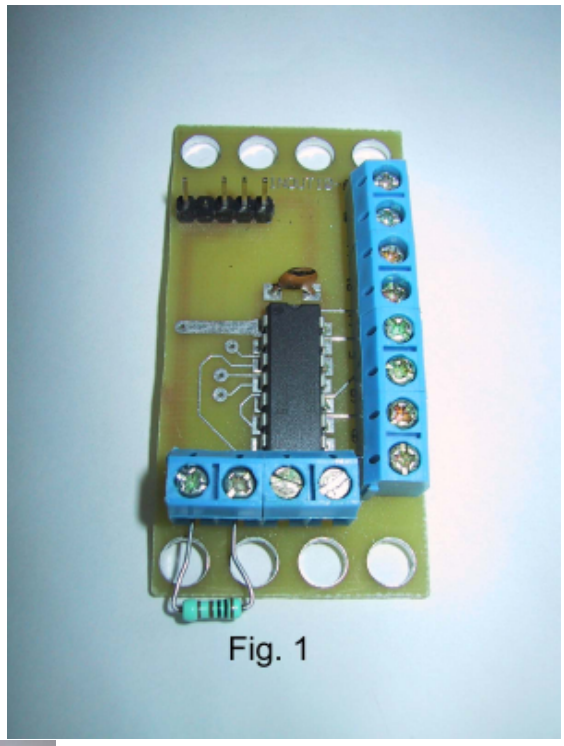


Fig. 1

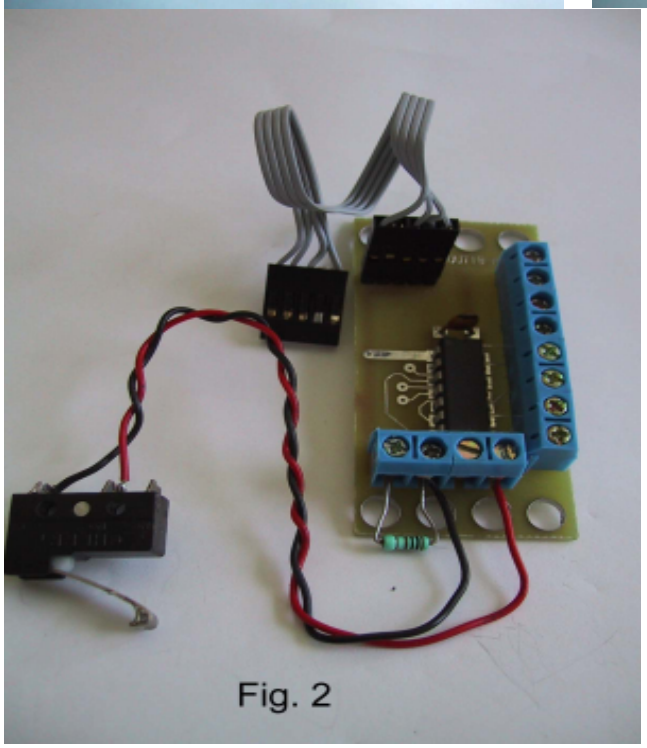


Fig. 2

Connecting a bump sensor to the InOut10 RoboBrick is quite simple. In this example we will connect only the left microswitch to the InOut10 brick so that each of the connections can be plainly seen. To add the right microsensor, follow the same instructions except use Terminal Pin 2 (input bit 9) instead of Terminal Pin 3 (input bit 8). The left and right microsensors share all other connections.

1. To begin, connect a 1 K ohm resister between Terminal Pins 3 and 4 of Terminal Block N3 as shown in Fig. 1 above. Terminal Pin 3 is input bit 8 and Terminal Pin 4 is Ground.

2. Make sure that the wires leading from the microswitch are shorted together when the microswitch lever is pressed..
3. Next, connect the microswitch between Terminal Pin 3 and Terminal Pin 1 (+5 v) of N3 and then plug−in the 4−wire cable as shown in Fig. 2.
4. Connect the other end of the 4−wire cable to the BS2Hub8 at socket N1

Programming Chart Excerpts

Command	Send/ Receive	Byte Value							Discussion
		7	6	5	4	3	2	1	0	
Read Inputs Low	Send	0	0	0	0	0	0	0	0	Return low order 5−bits of input *iiiii* (after XOR'ing with complement mask)
	Receive	0	0	0	*i*	*i*	*i*	*i*	*i*	
Read Inputs High	**Send**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**Return high order 5−bits of input *IIIII* (after XOR'ing with complement mask)**
	Receive	**0**	**0**	**0**	***I***	***I***	***I***	***I***	***I***	
Read Complement Mask Low	Send	0	0	0	0	0	0	1	0	Return low order 5−bits of complement mask *ccccc*

Bump Sensor Software

To get the status of the bump sensor we need only to write a short procedure. The first step is to determine what command the BS2Hub8 needs to send to the InOut10 brick to get the status of its port connections where the bump sensors are connected (pins 8 and 9). Checking the InOut10 Programming Table we see that the command **Read Inputs High** will allow us to read the five high order connections (bits 5 − 9) if we send a byte value of 1 to the InOut10 brick. Once the InOut10 receives the **Read Inputs High** command, it will respond by sending back a byte that represents the values appearing on the high order five bits. The value sent will be saved in the variable "inout " and since we will set the complement bit to zero, the complementing done automatically by the InOut10 Brick will not change its value (setting the complement byte equal to $ff would have converted an active low output to an active high or alternatively and active high output to an active low − another nice feature to have available).

PSEUDO CODE

```
initialize variables
loop
        send read command
        receive result
        process result
loop end
```

BASIC STAMP II − TEST CODE

```
'Initializations
baud_code con 396               'BS2 uses 396;BS2SX uses 1021 for 2400 baud
        dirs = %1111111101111111        'Set the BS2 bit directions (0 = input; 1 = output)
        inout var byte                  'Define a variable to hold the bump sensor information
        inout10_in con 8                'Serial data in on the BS2Hub8 at socket N1 (stamp pin 8)
        inout10_out con 9               'Serial data out on the BS2Hub8 at socket N1 (stamp pin 9
        high inout10_out                'This prepares the output data line for the start of seri
        sensor_Mask con $18             'Mask off all but the bump sensor bits

Main:
        gosub Bump
        inout, [Forward, Escape_Left, Escape_Right, Escape_Left]        'Branch arguments are dir
        goto Main
```

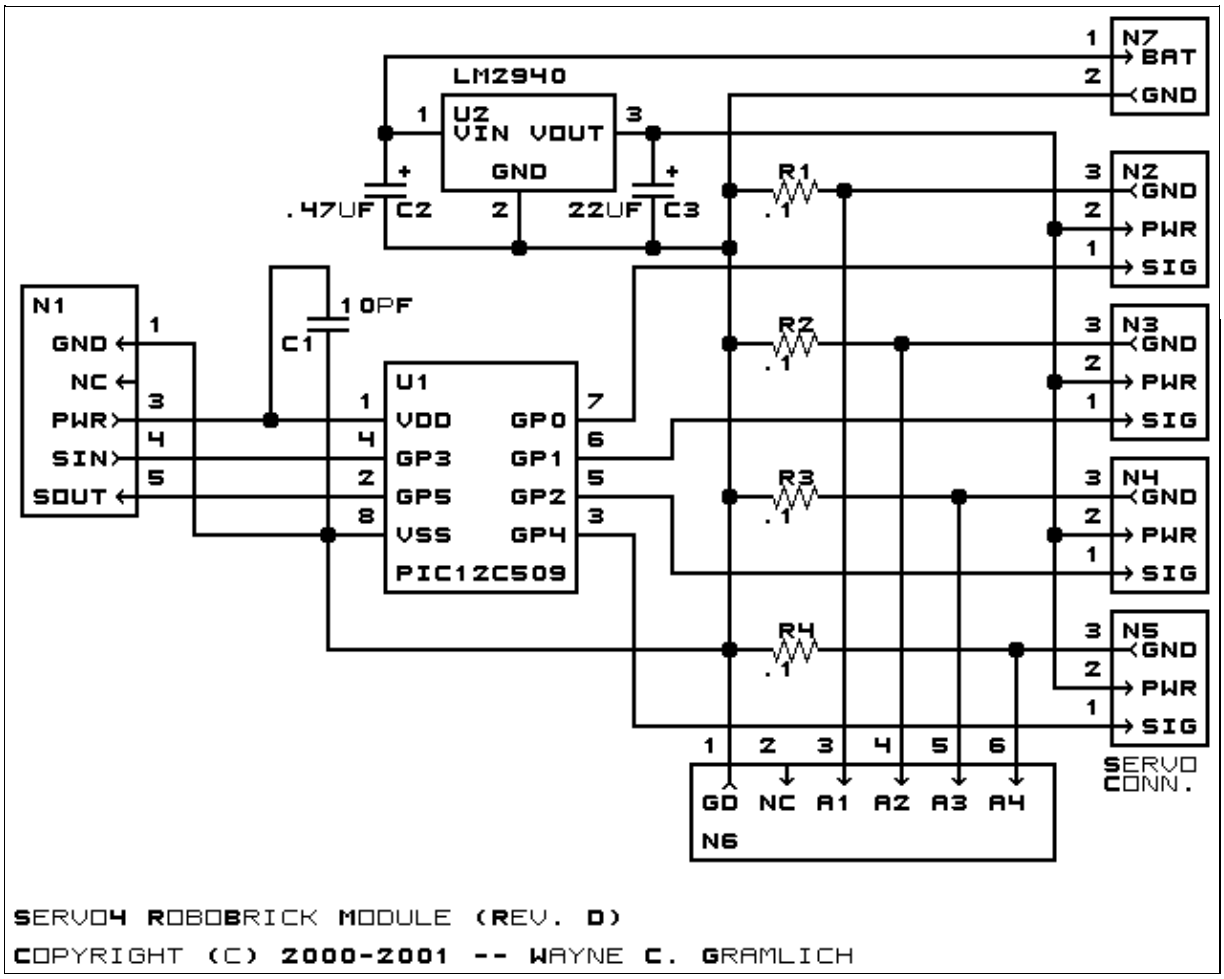
```
'Bump Sensor Subprocedures

Bump:
        serout inout10_out, baud_code, [16]                'Read Inputs High − Send
        serin inout10_in, baud_code, [inout]               'Read Inputs High − Receive  (high order 5 bits in variab
        inout = inout & sensor_mask                        'Mask off all but the bump sensor bits
        inout = inout >> 3                                 'Shift Bump Sensor bits to positions 0 and 1
        return

Foward:
        debug "No Bump − Going Forward",cr
        return

EscapeLeft:
        debug "Bumped on Right − Escaping Left", cr
        return

EscapeRight:
        debug "Bumped on Left − Escaping Right",cr
        return
```

The Servos

The robot we are building will use differential steering provided by two continuously rotating servo motors. Modifying a standard servo so that it can rotate continuously will not be covered here but information on making this modification can be found at Servo Mod on the Seattle Robotics Society website. We will program each servo with four responses: stop, full reverse, full forward and slow forward. By combining the responses appearing on each of the two servo motors, we can create a variety of robot motions: halt, go forward, go in reverse, turn left, turn right, pivot left and pivot right. The Servo4 brick that we will be using is dedicated exclusively to servo operation and it can accommodate up to four individual servos each of which can be independently controlled by the user.

The Servo4 brick provides control the speed/direction of each servos' with 8−bits of resolution. For example, the Set High command sets the high order 4−bits and the Set Low command sets the low order 4−bits. Now keeping in mind that a Servo4 brick can control up to 4 servos, let's take a look, for a moment, at the Servo4 Programming Table to see how to address a specific servo. To set all of the high order bits high on servo 1 (servos are numbered from 0 − 3), the table tells us that the Set High command byte we must send to the Servo4 board is 00111101 = $3D. The last two bits of this command byte are used to signal which servo is the one designated to receive the command.

Servo4 Schematic



The Servo4 RoboBrick will accomodate 4 servo motors but unlike most of the other RoboBricks, the servos are not powered by the selected HUB. A separate power source connection is provided on the Brick. Each servo plugs directly into a 3−pin header with pin configurations as shown in the diagram above. We will plug our the left servo into header 0 and the right servo into header 1. The last thing to do is connect a 4−line cable from the Servo4 brick to the BS2Hub8 brick at socket N2 (input − pin 10, output − pin 11). The following table lists a number of available servos together with lead and color identification to help insure that proper connection is made to the Servo4 brick.

Programming Chart Excerpts

Command	Send/ Receive	Byte Value								Discussion
		7	6	5	4	3	2	1	0	
Set High	**Send**	**0**	**0**	***h***	***h***	***h***	***h***	***s***	***s***	**Set high order 4 bits of servo *ss* to *hhhh* and set the remaining 4 low order bits to zero**
Set Low	**Send**	**0**	**1**	***l***	***l***	***l***	***l***	***s***	***s***	**Set the low order 4 bits of servo *ss* position to *llll*.**
Increment	Send	1	0	0	*i*	*i*	*i*	*s*	*s*	Add *iii* to the position of servo *ss*.
Decrement	Send	1	0	1	*d*	*d*	*d*	*s*	*s*	Subtract *ddd* from the position of servo *ss*.
Read Enable	Send	1	1	0	1	0	1	*s*	*s*	Return the enable bit *e* for servo *ss*.
Set Enables	**Send**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**Set enable flags for all four servos to *eeee*.**

Send	**0**	**0**	**0**	**0**	***e***	***e***	***e***	***e***

Servo4 Software

We will need to create separate procedures to translate the individual servo responses into each of the desired robot motions listed above and, in addition, we will need to add a few lines of code in the Initializations section to tie everything together. A program note: servos can be either enabled or disabled (see the Servo4 Programming Table) so we must insert some code to insure that our two servos are enabled for use. The Set Enables command requires two command bytes to be sent consecutively. With these two command bytes, each of the four servos can be individually enabled or disabled and therefore 16 different configurations are possible when all four servos are considered together.

PSEUDO CODE

PROGRAM CODE

Note that the most of the code consists of repeatedly sending simple command bytes to the servo board. Conspicuously absent are the servo timing loops and the servo pulse width specifications. These are handled by the Servo4 brick so that the user does not need to concern himself with the complication of those details. We are at liberty therefore to ignore the servos so long as we are not changing either their speed or direction.

```
'Initializations
        baud_code con 396
        dirs %1111110111111111
        servo4_in con 10                        'Serial data "in" on the BS2Hub8 at socket N2 (st
        servo4_out con 11                       'Serial data "out" on the BS2Hub8 at socket N2 (s
        high servo4_out                         'This prepares the output data line to start seri
        enable_a con $d9    Set Enables command '
        enable_b con $03                        'bytes to enable servos 0 and 1 and disable servo
        serout servo4_out, baud_code, [enable_a] 'First enable all servos command
        serout servo4_out, baud_code, [enable_b] 'Second enable all servos command

'Servo direction assignments
        l_halt_hi con $24                       'The numberical values for each motor response wi
        l_halt_lo con $44                       'specific servos used
        r_halt_hi con $25
        r_halt_lo con $45
        l_fwd     con $3d                       'When the servo high bits are all set to 1, that
        r_fwd     con $00                       'ignore any value set on the servo's low bits
        l_bak     con $01
        r_bak     con $3c
        l_slo     con $24
        r_slo     con $25

'Servo4 subprocedures
Halt:
        serout servo4_out, baud_code, [l_halt_hi]
        serout servo4_out, baud_code, [r_halt_lo]
        serout servo4_out, baud_code, [l_halt_hi]
        serout servo4_out, baud_code, [r_halt_lo]
        return

Forward:
        serout servo4_out, baud_code, [l_fwd]
        serout servo4_out, baud_code, [r_fwd]
        return

Backup:
        serout servo4_out, baud_code, [l_bak]
```

```
        serout servo4_out, baud_code, [r_bak]
        return

L_Turn:
        serout servo4_out, baud_code, [l_slo]
        serout servo4_out, baud_code, [r_fwd]
        return

R_Turn:
        serout servo4_out, baud_code, [l_fwd]
        serout servo4_out, baud_code, [r_slo_hi]
        serout servo4_out, baud_code, [r_slo_lo]
        return

L_Pivot:
        serout servo4_out, baud_code, [l_halt]
        serout servo4_out, baud_code, [r_fwd]
        return

R_Pivot:
        serout servo4_out, baud_code, [l_fwd]
        serout servo4_out, baud_code, [r_halt]
        return

Escape_Left:
        gosub Backup
        pause 1000
        gosub L_Pivot
        pause 500
        return

Escape_Right:
        gosub Backup
        pause 1000
        gosub R_Pivot
        pause 500
        return
```

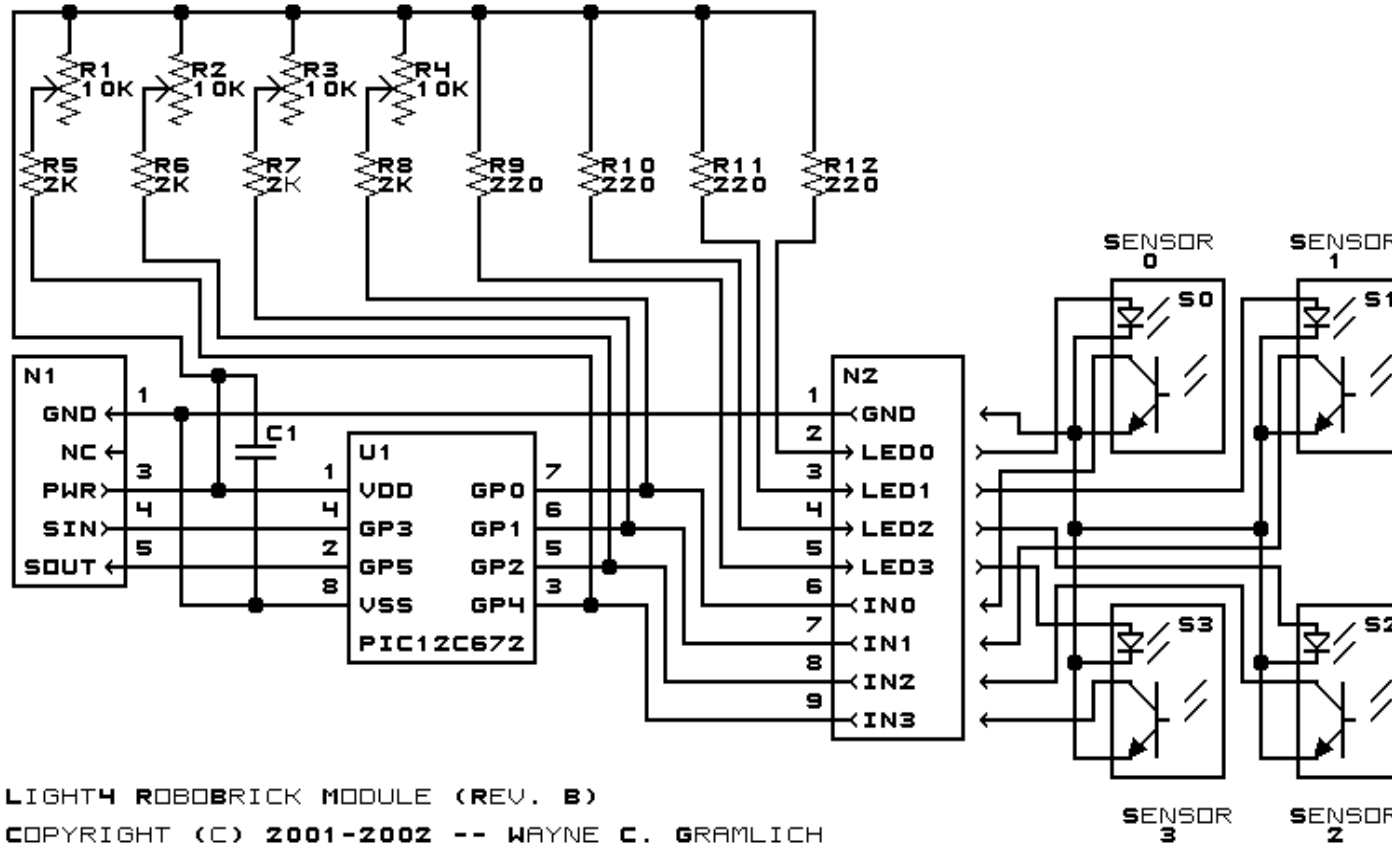
Table Edge Detectors

Since we are developing a tabletop robot, we need some way to detect and avoid the table's edge so that our robot won't accidentally plummet to the floor and thus become a pile of parts no longer recognizable as a robot. To do this, we have selected the Infrared Sensor package shown in the figure below. This sensor has an infrared light source provided by an IR LED packaged together with an infrared detector. For our particular use, the LED will be set to a steady "on" state. The detector continuously measures the amount of light being reflected from the surface facing the sensor, in this case, the table top. When the sensor is peering over the edge, there will be a dramatic drop in the detectors output as the warning that we must take some immediate action to avoid falling off the table. We will install one, downward looking Sensor Package on each of the four corners of our robot to give it the as much protection as possible.

The detector, used in these sensors, produces an analog output that we will need to convert to a digital signal. The Light4 RoboBrick was designed precisely with this need in mind and so we will use it to construct our Table Edge Detectors. The Light4 RoboBrick will perform the needed analog−to−digital conversion and it will output an 8−bit digital number that is a relative measure of the intensity of the IR light reflected back to the detector. Let's take a look a the Light4 Programming Table to see what commands we'll need to accomplish our purpose. We can check the status of sensor number 0, 1, 2 or 3 by sending the command "00", "01", "02", or "03" respectively. We will receive back from the Brick, an 8−bit number representing

the input value of the sensor queried. The Light4 board will be connected to the BS2Hub8 at socket 3 (pins 13 and 14).

Light4 Connections



Connecting the Sensor Packages to this board requires nothing more that attaching each of the sensor's connectors to the appropriate post on the Light4 terminal blocks. The photos below show how to attach one of the sensors. Again for clarity, we have used only one sensor; attaching the other three follows the same simple approach. The sensors each have dedicated connections for their LEDs and Detector outputs but they all share the +5 volt power source and ground .

Programming Chart Exerpts

Command	Send/ Receive	Byte Value								Discussion
		7	6	5	4	3	2	1	0	
Read Pin	**Send**	**0**	**0**	**0**	**1**	**0**	**0**	***b***	***b***	**Read pin bb and respond with 8−bit value vvvvvvvv**
	Receive	***v***	***v***	***v***	***v***	***v***	***v***	***v***	***v***	
Read Low Threshold	Send	0	0	0	1	0	1	*b*	*b*	Return low threshold for pin *bb* of *llllllll*
	Receive	*l*	*l*	*l*	*l*	*l*	*l*	*l*	*l*	
Set High Threshold	**Send**	**0**	**0**	**0**	**1**	**1**	**0**	***b***	***b***	**Set high threshold for pin *bb* to *hhhhhhhh***
	Send	***h***	***h***	***h***	***h***	***h***	***h***	***h***	***h***	
Set Low Threshold	Send	0	0	0	1	1	1	*b*	*b*	Set low threshold for pin *bb* to *llllllll*
	Send	*l*	*l*	*l*	*l*	*l*	*l*	*l*	*l*	

Edge Detector Code

PSEUDO CODE

PROGRAM CODE

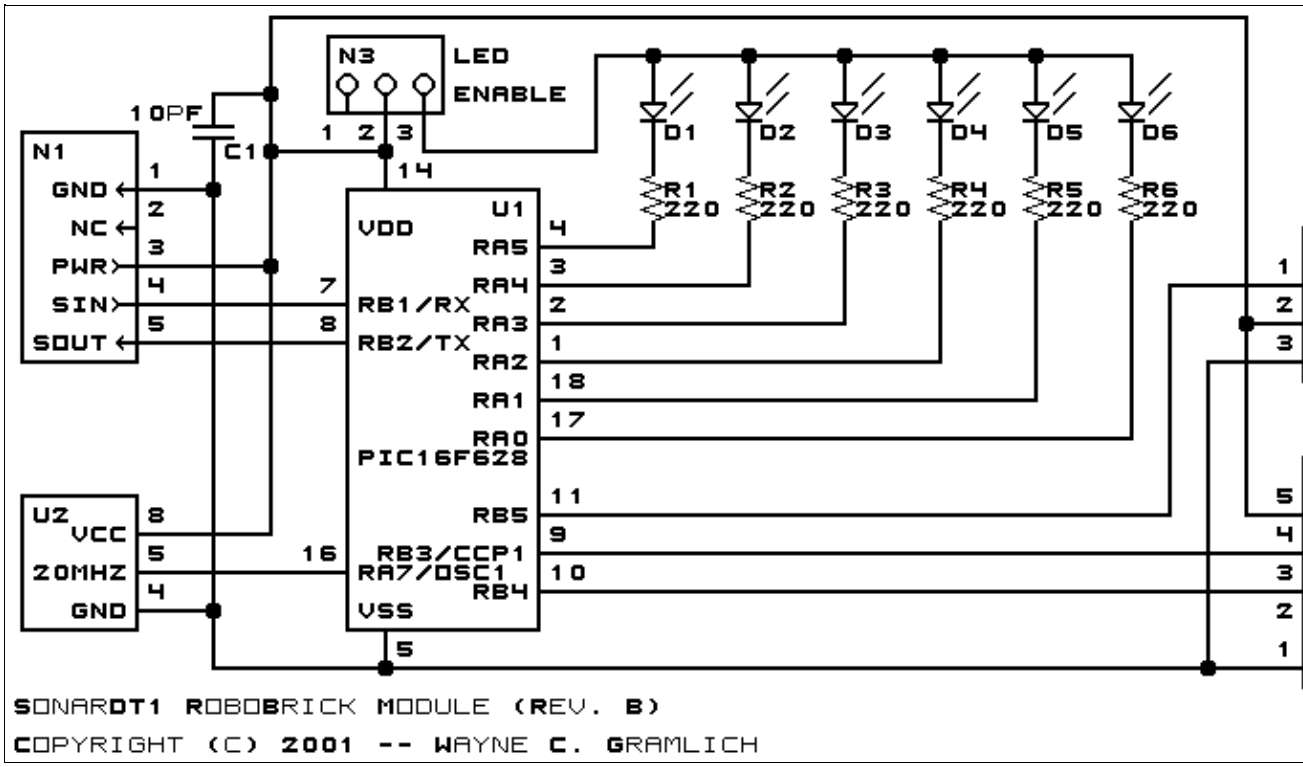
```
'Initializations

        baud_code con 396          'BS2 code for 2400 baud
        dirs = %110111111111111    'pin 13 is an input, all other pins are output

        edge_out con 14
        edge_in  con 13
        high edge_out

        edge0       var byte
        edge1       var byte
        edge2       var byte
        edge3       var byte
        front_edge  var byte
        back_edge   var byte
        threshold   var byte

'Using default high and low thresholds of $c0 and $40

Main:
        gosub find_edge
        branch front_edge, [RearSensor, r_turn, l_turn, back_up]

RearSensor:
        branch back_edge, [do_nothing, r_turn, l_turn, forward]
        pause 3000
        goto Main

'Procedures:

Do_Nothing:
        debug "Do Nothing", cr
        return

R_Turn:
        debug "Turn right − front left sensor or rear left sensor", cr
        return

L_Turn:
        debug "Turn left − front right sensor or rear right sensor", cr
        return

Back_Up:
        debug "Back Up − a rear sensor", cr
        return

Forward:
        debug "Forward − both rear sensors are active", cr
        return

Find_Edge:

'Left Front Detector
        serout edge_out, baud_code, [4]      'Read Pin − Check left front detector
        serin edge_in, baud_code, [edge]    'Value returned is 0 or 1
```

```
return
```

Object Detector

In this application, we have chosen to use the DTSonar as our object avoidance sensor. The DTsonar brick controls both the DTSonar board and a dedicated servo that can be used to mechanically scan the servo in an arch of about 160 degrees. This give us the ability to obtain both bearing (servo position) and distance (sonar measurement) information about objects in the vicinity of the robot. For this particular example, we will establish three positions for the servo: 45 degrees left of center, center, and 45 degrees right of center. At each servo position we will use sonar to determine the distance to the closet object. The robot will respond by turning away from any "near" object (one that is closer than ??? inches ahead). If a near object is on the left, the robot will turn right: if directly ahead, the robot is arbitrarily designed to turn left; etc. If the robot is presented with two or more near objects, it will do a 90 degree course reversal.

DTSonar Connections



Programming Chart Exerpts

Command	Send/ Receive	Byte Value								Discussion
		7	6	5	4	3	2	1	0	
Read Distance Low	Send	0	0	0	0	0	0	0	0	

	Receive	*l*	*l*	*l*	*l*	*l*	*l*	*l*	*l*	Return the low order byte *llllllll* of the distance
Read Distance High	Send	0	0	0	0	0	0	0	1	Return the high order byte *hhhhhhhh* of the distance
	Receive	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	
Read Distance High and Low	Send	0	0	0	0	0	0	1	0	Return the low and high order bytes *llllllll hhhhhhhh* of the distance
	Receive	*l*	*l*	*l*	*l*	*l*	*l*	*l*	*l*	
	Receive	*h*	*h*	*h*	*h*	*h*	*h*	*h*	*h*	
Trigger Distance Measure	**Send**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**Trigger a Single Distance Measurement**
Disable Servo	Send	0	0	0	0	0	1	0	0	Disable Servo
Enable Servo	**Send**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**Enable Servo**
Disable Continuous Measurement	Send	0	0	0	0	0	1	1	0	Disable Continuous Measurement
Enable Continuous Measurement	Send	0	0	0	0	0	1	1	1	Enable Continuous Measurement
Increment Servo	Send	0	0	0	0	1	0	0	0	Increment servo position by one.
Decrement Servo	Send	0	0	0	0	1	0	0	1	Decrement servo positon by one.
Read Servo	Send	0	0	0	0	1	0	1	0	Return servo value *ssssssss* of the distance
	Receive	*s*	*s*	*s*	*s*	*s*	*s*	*s*	*s*	
Read Enables	Send	0	0	0	0	1	0	1	1	Return servo enable *s* and continuous distance measurement *m*
	Receive	0	0	0	0	0	0	*m*	*s*	
Set Servo Low	Send	0	0	0	1	*l*	*l*	*l*	*l*	Set the low order 4 bits of the servo position to *llll*.
Set Servo High	**Send**	**0**	**0**	**1**	**0**	***h***	***h***	***h***	***h***	**Set the high order 4 bits of the servo position to *hhhh*.**

DTSonar Code

PSEUDO CODE

PROGRAM CODE

Development & Debugging Bricks

1. **Descriptions & function for each module**
2. **Some typical development uses**
3. **Some typical debugging uses**