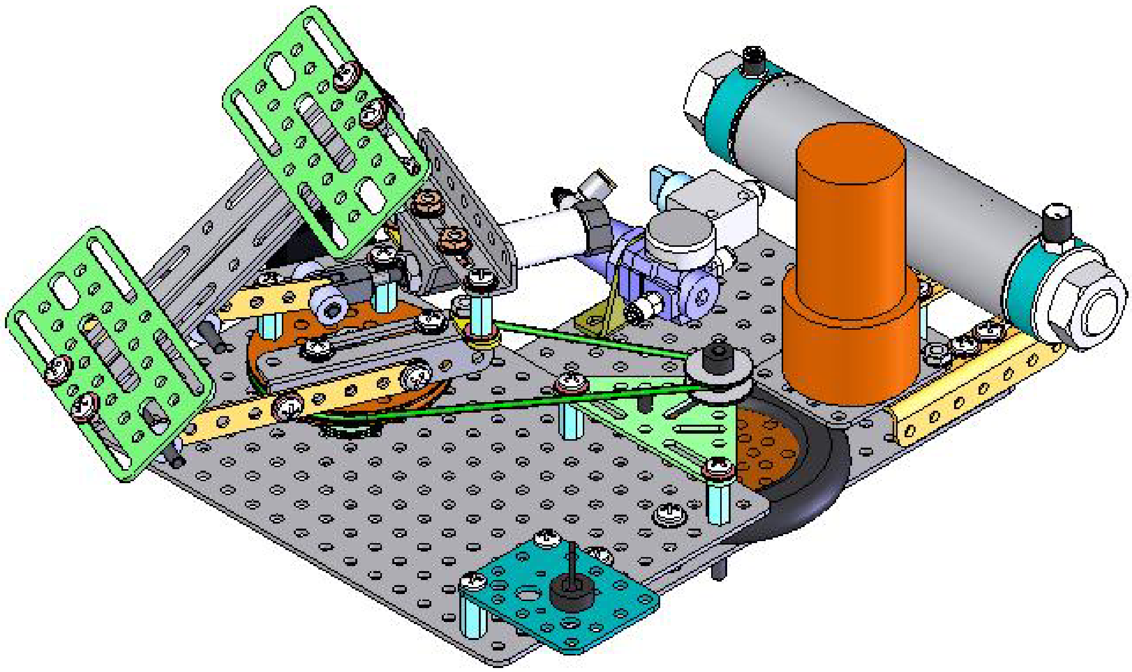
Solar Tracker Construction Guide

An Illustrated Assembly Manual



The GEARS solar tracking assembly instructions are organized into three (3) sections:

Section One: Information necessary to complete the mechanical assembly of the project as well as instructions for the placement of the pneumatic components.
Section Two: Detailed Instructions on how to configure and operate the pneumatic components.
Section Three*: Wiring and programming instructions for integrating the Basic Stamp© two axis controller.

The suggested control system for this project should include a microprocessor fitted with sensors to track the sun's position. We recommend two products:
1.) The Parallax Basic Stamp (www.parallax.com)
2.) Machine Science's (www.machinescience.com) C based control system

The tracking mechanism can be fitted with several different energy transformation systems including a parabolic reflector *(High temperature solar heating device)*, a flat plate water or air heater, or a photovoltaic array used to charge the battery that operates the device. The choice of which renewable energy system to use is left to the instructor and students.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Section One: Mechanical Assembly

Required Tools

Safety Glasses
Phillips Head Screwdriver
Allen Wrench or Hex Key (sizes .050, 1/16, 5/64, 3/32, and 1/8)
Wrench (sizes 3/8 and 9/16)
Wire Strippers and Crimpers
Needle Nose Pliers
Tubing cutter or Shears
Matches or Lighter (To melt the polycord)
Smooth File (Small)

Materials

Structural

Qty.	
2	6 x 9 Plate
5	7 Hole Angles
3	13 Hole Angle
1	5 Hole Flat Bar
2	9 Hole Flat Bar
2	180 Degree Fish Plates
1	IM15 Motor Mount
1	Switch Plate
2	Bearing Plates
3	3/16” x 4” axle
2	3/16” dia. x 1-1/2” axle
2	Hex Adapter 3/16” Bore
2	3” Hex Wheel
1	Tire
1	Sine Triangle
1	1/8” dia. x 20” Green Polycord

Hardware

Qty.	
55	#10 Flat Washers
53	#10 Lock Washers
7	#10 Fender Washers
5	½” Flat Washers
38	#10-24 x 3/8” Machine Screws
2	#10-32 x 3/8” Machine Screws (Motor)
36	#10-24 x 1/2” Machine Screws
18	#10-24 Coupling nuts or Standoffs
16	#10-24 Hex Nuts
13	3/16” ID Shaft Collars

Electrical

2 ea.	Wire Nuts (grey and blue)
4	¼” Female Quick Disconnects
2	¼” Male Quick Disconnects
2	0,201” Female Quick Disconnects
8”	16 ga Insulated Wire (blk and red)
1	Motor
1	Electronic Speed Controller
1	SPST Toggle Switch
1	Battery
1	Machine Science/ Parallax Microprocessor Kit
	Sensors as needed

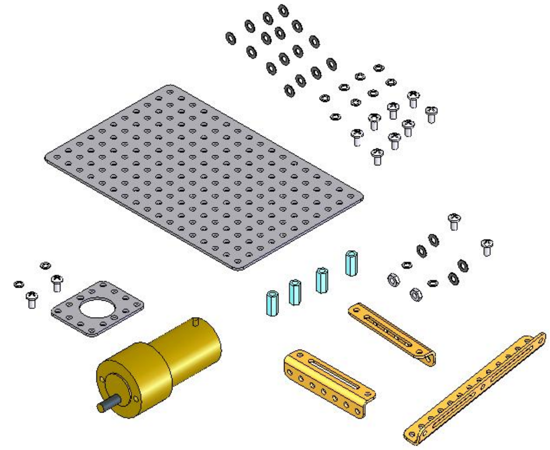
Misc.

Qty.
1 1” Length of Surgical Tubing
8 8” zip ties

Pneumatics

Qty.
1 3-2 Solenoid Valve
1 3-2 Manual Valve
1 Reservoir
1 Linear Actuator (Cylinder)
1 Regulator with Pressure Gauge
4’ 4mm tubing
1 Pneumatic Bracket

Assemble the Motor Mount Module



Step One: Construct the Adjustable Motor Assembly

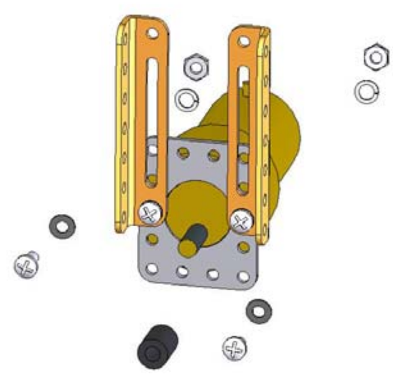
<u>Necessary Components</u>

Qty.	Description
20	#10 Flat Washers
12	#10 Lock Washers
10	#10-24 x 3/8” Machine Screws
2	#10-32 x 3/8” Machine Screws (Motor)
4	#10-24 Coupling Nuts
1	6 x 9 Plate
2	7 Hole Angles
1	13 Hole Angle
1	6 x 9 plate
1	IM15 Motor Mount
1	Motor
1	1” Length of Surgical Tube (Not Shown)

Step One: Assemble the Motor and Motor Mount

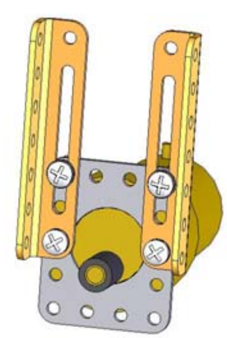
<u>Components</u>

4	#10 lock washers	2	#10-32 x 1/2” machine screws
2	#10-24 hex nuts	2	7 hole angles
2	#10 Flat washers	1	Motor mount
2	#10-24 x 3/8” machine screws	1	Motor
		1	¼” id x .7” surgical tubing



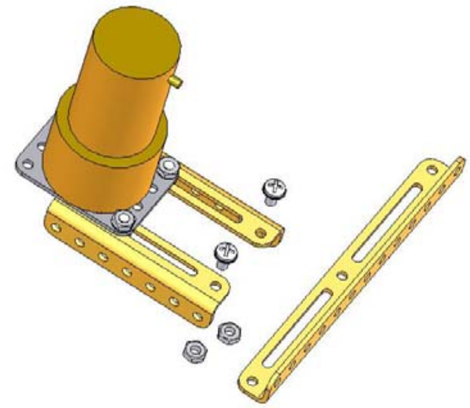
Procedure

1. Attach the motor mount and two 7 hole angles to the motor using two #10-32 machine screws and lock washers.
2. Use 2 #10-24 x 3/8” machine screws, flat washers, lock washers and nuts to secure the 7 hole angles to the top-most motor mount holes. (Note: See illustration on bottom right of the page)
3. Measure the motor shaft and cut the surgical tubing 1/16” shorter than the motor shaft.
4. Slide the surgical tubing onto the motor shaft as shown in the bottom right illustration. Keep the surgical tubing just off the motor face so it will not rub on the motor face or motor mount.



105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 78

Step Two: Position the Motor Mount Module

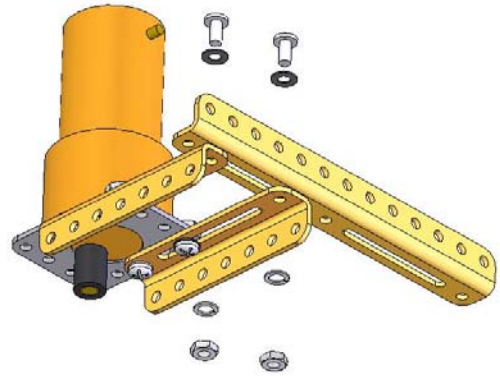


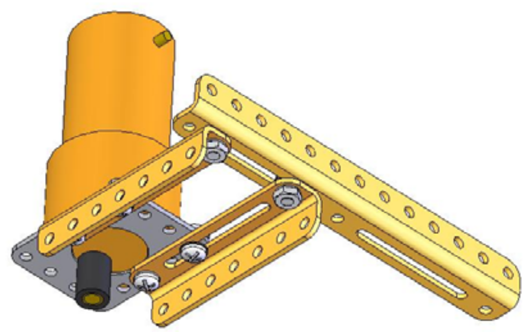
Necessary Components

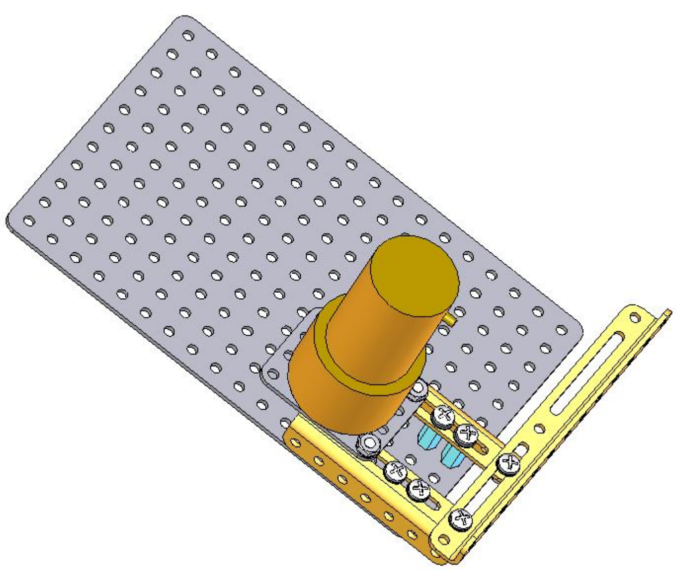
Qty.	Description
10	#10 Flat Washers
10	#10 Lock Washers
10	#10-24 x 1/2" Machine Screws
4	#10-24 Coupling Nuts
1	13 Hole Angle
1	6 x 9 Plate
1	Motor Mount Module Pre-assembled
2	#10-24 Hex Nuts

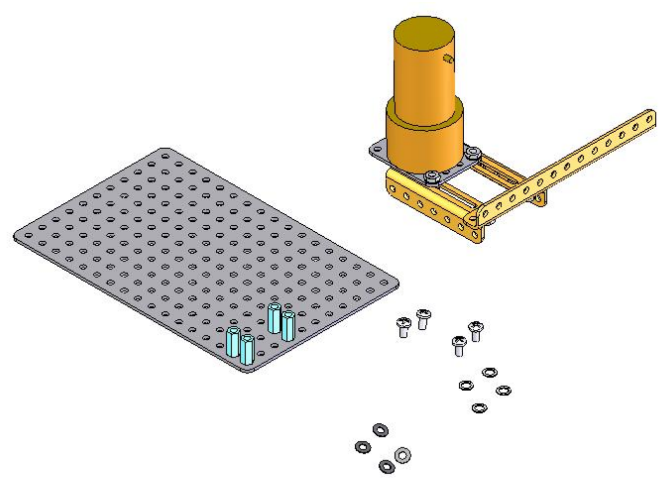
Procedure

1. Attach the 13 hole angle to the motor mount assembly as shown.
2. Position the #10-24 coupling nuts on the 6 x 9 plate as shown below. Be certain to count the holes and position the coupling nuts exactly as shown.
3. Fasten the motor mount module onto the 6 x 9 plate. Do not tighten the mounting bolts at this time. The motor mount module should slide easily back and forth along the tops of the coupling nuts. This will be necessary in order to position the friction wheel in the next sequence.

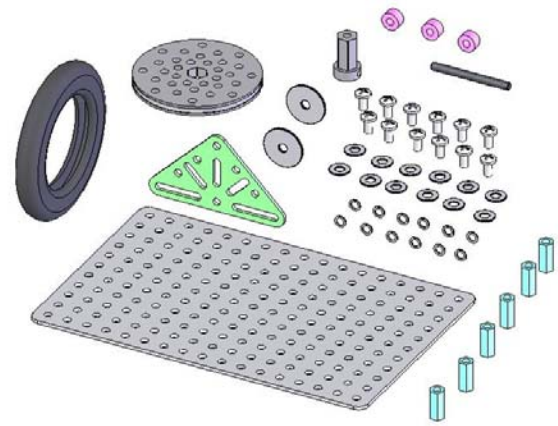






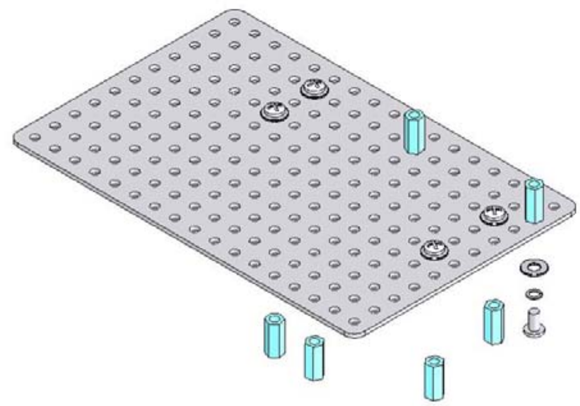


Friction Wheel Module



Necessary Components

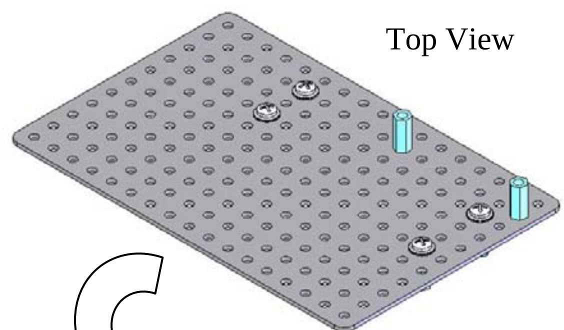
Qty.	Description
1	4" x 3/16" axle
3	3/16" ID Shaft Collars
1	Hex Adapter 3/16" Bore
1	3" Hex Wheel
1	Tire
8	#10 Flat Washers
8	#10 Lock Washers
2	#10 Fender Washers
8	#10-24 x 3/8" PH Machine Screws
6	#10-24 Coupling Nuts
1	Sine Triangle
1	6 x 9 Plate
1	Motor Mount Module (See preceding page)



Step One: Construct the Friction Wheel Mount

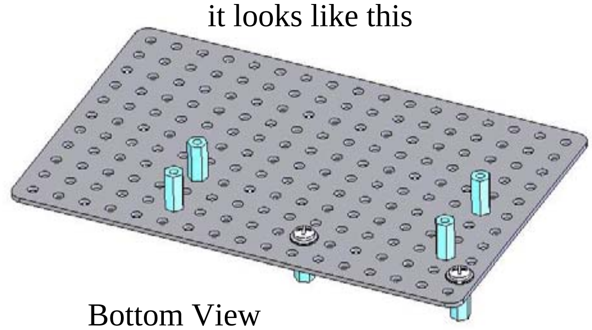
Procedure

1. Fasten (2) two #10-24 coupling nuts to the top side of the 6x9 plate. Use the #10-24 x 3/8" PH machine screws, lock washers and flat washers as shown in the illustration on the right.
2. Fasten (4) #10-24 coupling nuts to the bottom side of the 6x9 plate. Use the #10-24 x 3/8" PH machine screws, lock washers and flat washers as shown in the illustration on the right. Note: Look closely at the position of the coupling nuts and screw heads. Caution: Be certain to count the holes and position them exactly as shown in the illustration.
3. Attach the sine triangle to the (2) two coupling nuts on the top of the 6x9 plate as shown in the illustration below.

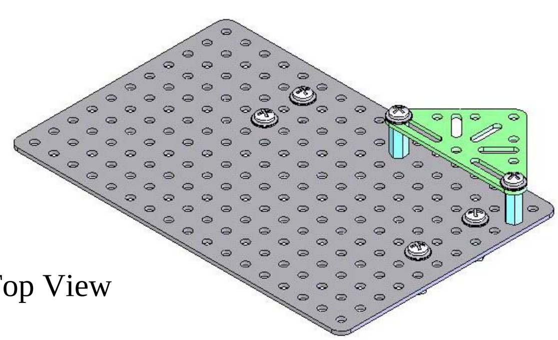


Top View

Flip the 6x9 plate over and it looks like this

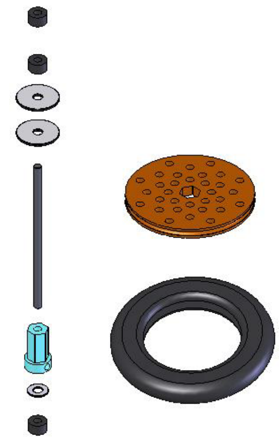


Bottom View



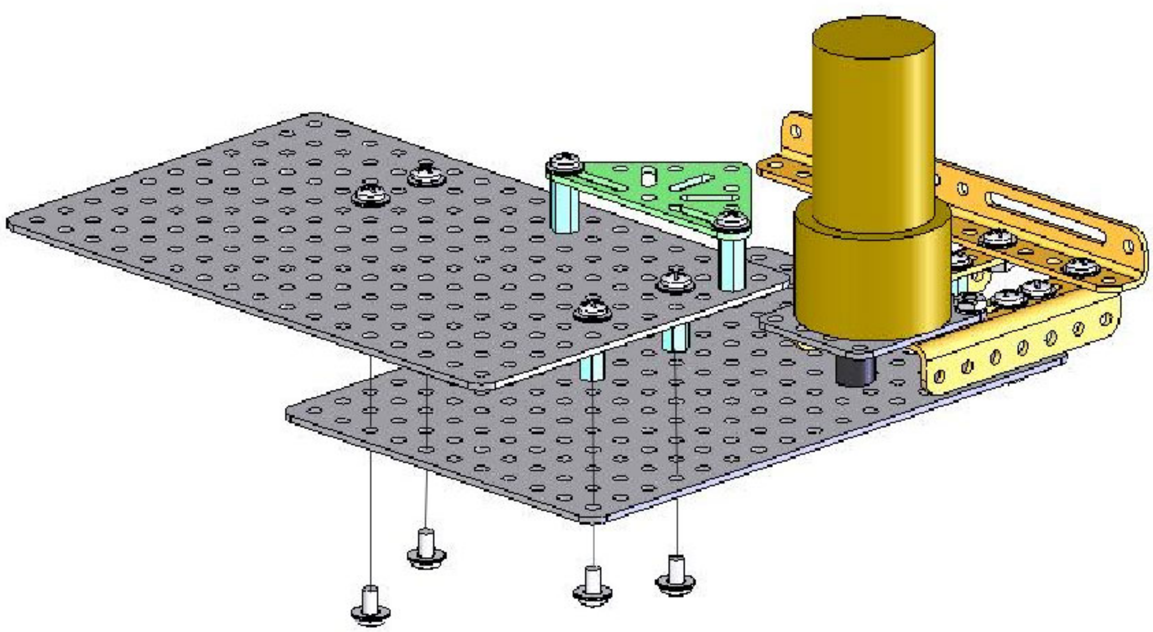
Top View

Step Two: Construct the Friction Wheel and Axle Assembly



Procedure

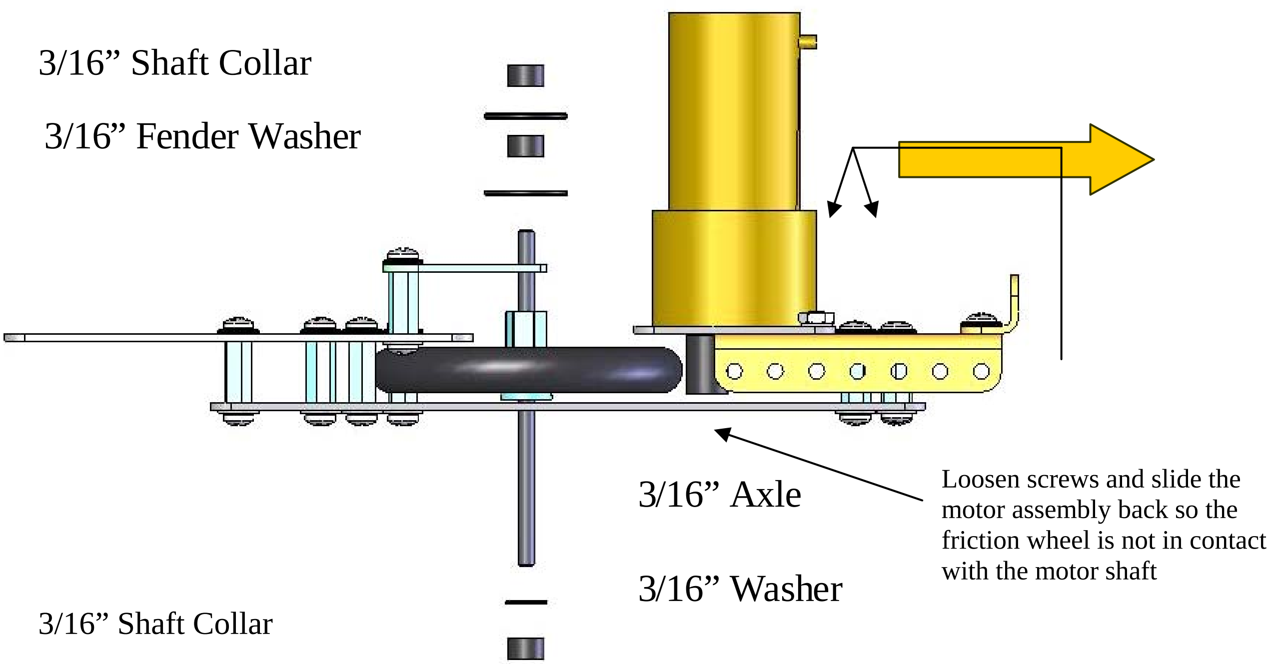
1. Collect the following parts:
   3pc. 3/16” shaft collars
   4pc. #10 Flat washer
   4pc. #10 – 24 x 3/8” PH Machine Screw
   4pc. #10 Lock Washer
   2pc #10 Fender washer
   1pc. 3/16” x 4” axle
   1pc. 3” Hex wheel
   1pc. Tire
   1pc. 3/16” Bore hex adapter
2. Note the alignment of the parts in the illustration on the top right.
3. Fit the tire to the 3” hex wheel as shown in the illustration on the right. The tire fits tightly onto the wheel and requires a bit of work to “Roll” it onto the wheel. Warming the tire in hot water can help. It also helps to use blunt tools like a spoon handle or a dulled flat screwdriver to help roll the tire onto the wheel.
4. Fasten the friction wheel mount to the (4) four coupling nuts on the bottom of the motor mounting plate as shown. Use (4) #10-24 x 3/8” PH machine screws, lock washers and flat washers. Caution: Be certain to count the holes and position them exactly as shown in the illustration.

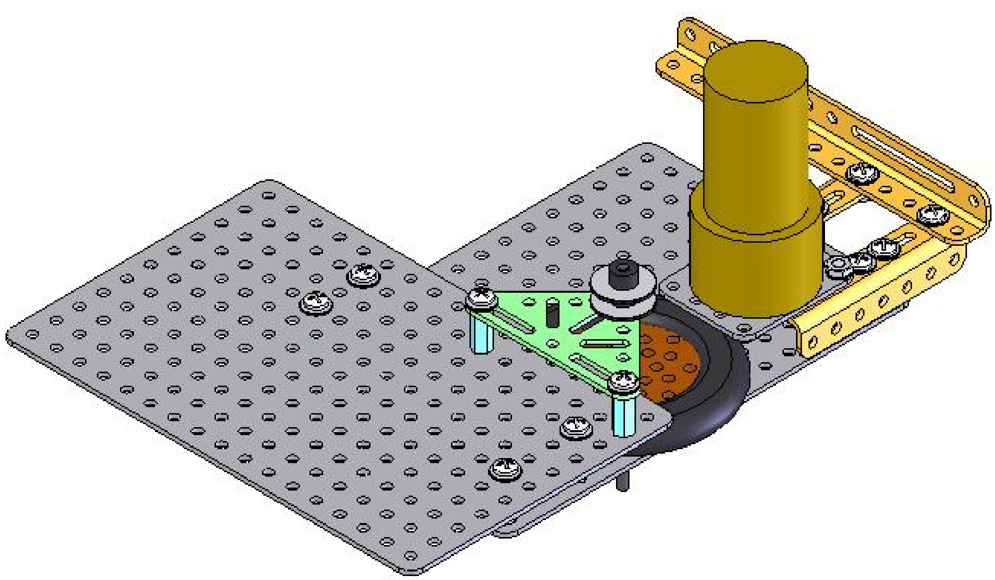


5. Assemble the hex adapter, wheel and tire. Capture this assembly by sliding the 3/16” axle up through the bottom of the motor mount plate and up through the sine triangle as shown below. It may be necessary to slide the motor mount assembly to the far right position in order to fit the wheel and tire assembly in place.

6. Fasten the remaining hardware (shaft collars, fender washers, and washer) in the order shown above. The finished friction wheel assembly is shown below. Note: The top of the axle should be flush with the top of the 3/16” shaft collar. Attach a battery to the motor leads, the drive assembly should turn freely.

Isometric View of Completed Friction Wheel Assembly





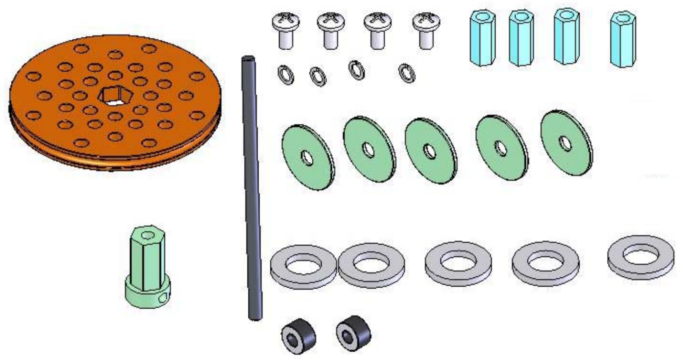
105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Altitude and Azimuth Positioning Module

A solar collector that follows the sun will need to rotate on an imaginary, east, west plane as well as a plane formed by looking up and down. The amount of rotation from east to west is measured in angles of "Azimuth" . The azimuth positioning module supports this east to west rotation. The amount of rotation, up and down is measured in angles of altitude. The altitude positioning module supports this up-down motion.

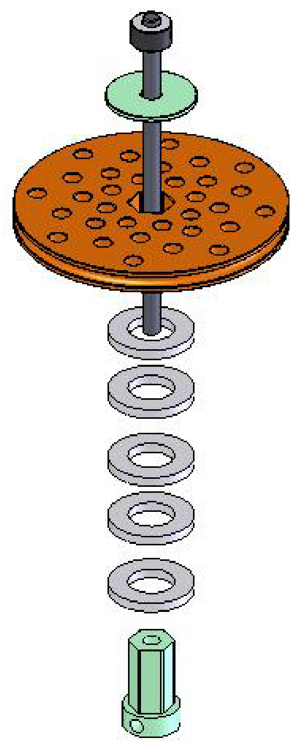
Step One: Construct the Azimuth Axis Assembly

Necessary Components

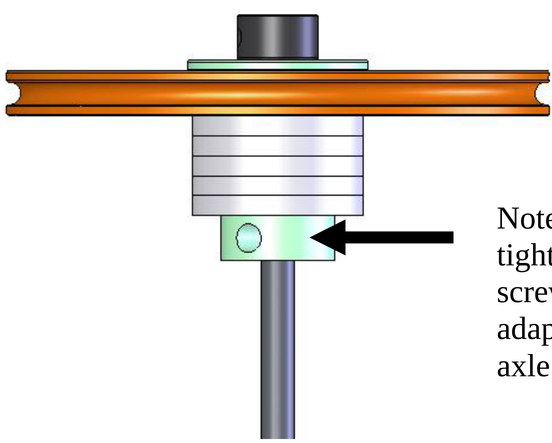


Qty.	Description
3-5	#10 Fender Washers
4	#10 Lock Washers
4	#10-24 x 1/2" Machine Screws
4	#10-24 Coupling Nuts
5	½" Flat Washers
2	3/16" Shaft Collars
1	3/16" dia. x 4" axle
1	3/16" Bore Hex Adapter
1	3" Hex Wheel
1	1/8" dia. x 16" Green Polycord (not shown)

Procedure



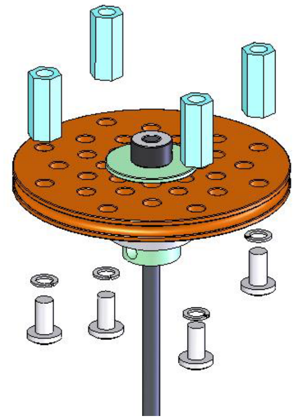
1. Collect and layout the parts listed above.
2. Load (5) five ½" washers onto the hex adapter as shown in the illustration on the right.
3. Slide the 3" hex wheel onto the hex adapter and washers.
4. Slide the 4" axle through the hex adapter bore.
5. Secure the 3" hex wheel onto the hex adapter using a 3/16" fender washer and a 3/16" shaft collar. The top of the axle should be flush with the top of the shaft collar. See the illustration below.

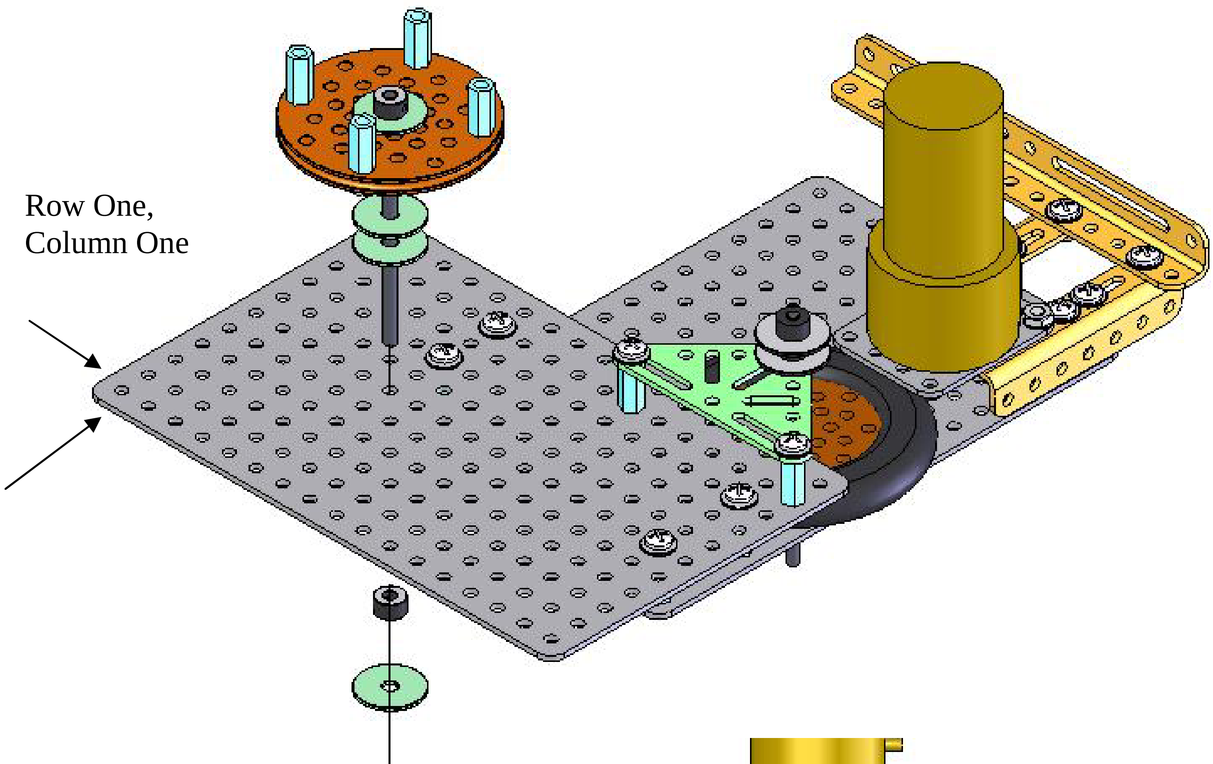


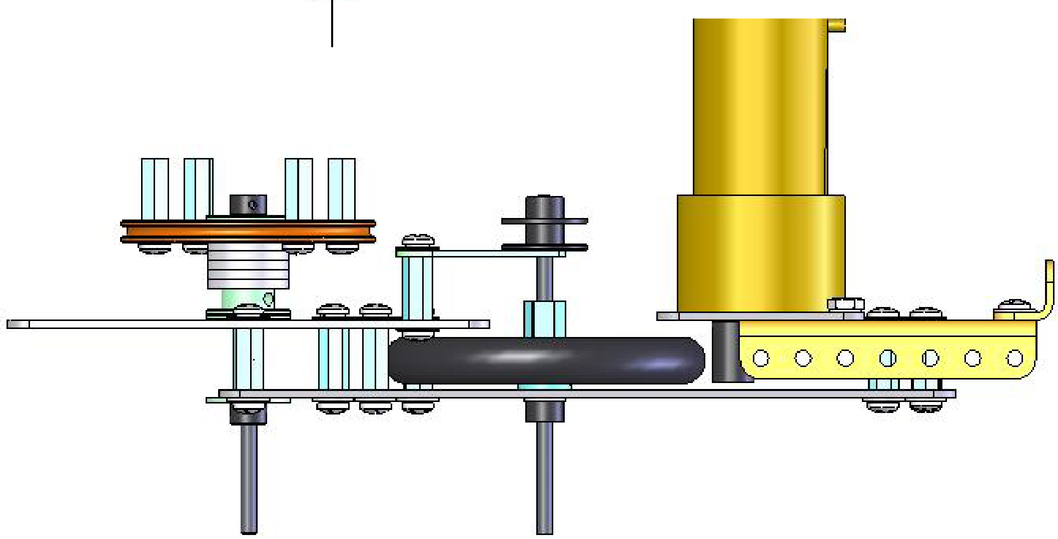
Note: It is not necessary to tighten the hex adapter set screw onto the axle. The hex adapter should turn freely on the axle.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

6. Fasten the (4) four coupling nuts to the 3” hex wheel using (4) four #10-24 x ½” ph machine screws and lock washers as shown.
7. Put (2) two 3/16” fender washers under the hex adapter and slide this assembly through both of the 6 x 9 plates on the Solar Tracker Module, as shown. The correct hole is located at the intersection of the 6th column and sixth row .
8. Secure this assembly to the solar tracking module using a #10 fender washer and a 3/16” shaft collar. The assembly should turn freely, but not be too loose. *Note: It is not necessary to over tighten set screws, over tightening will cause the hex key wrench to rotate in the screw socket and damage both the wrench and the set screw.*

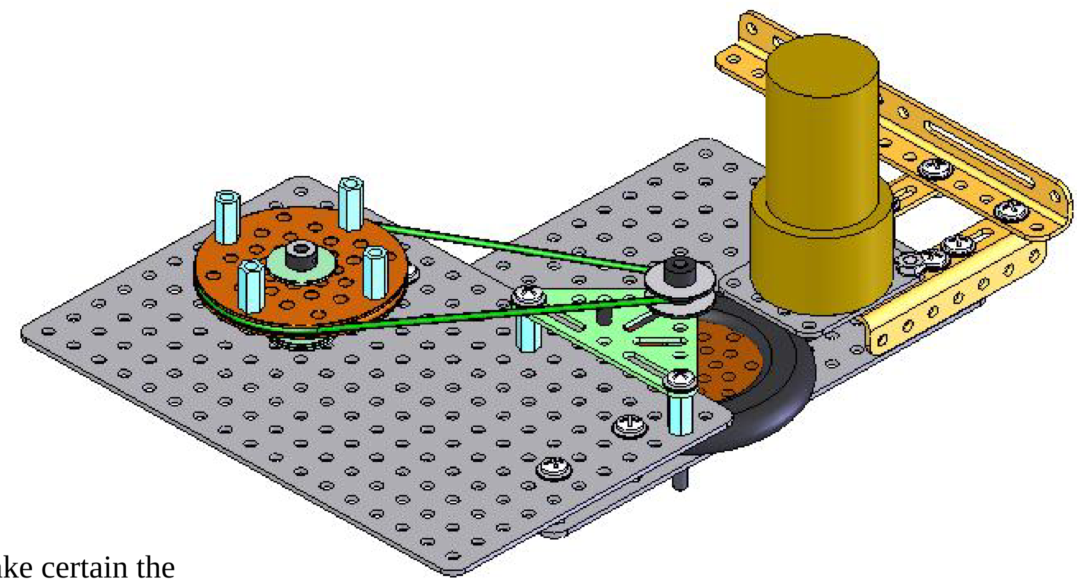


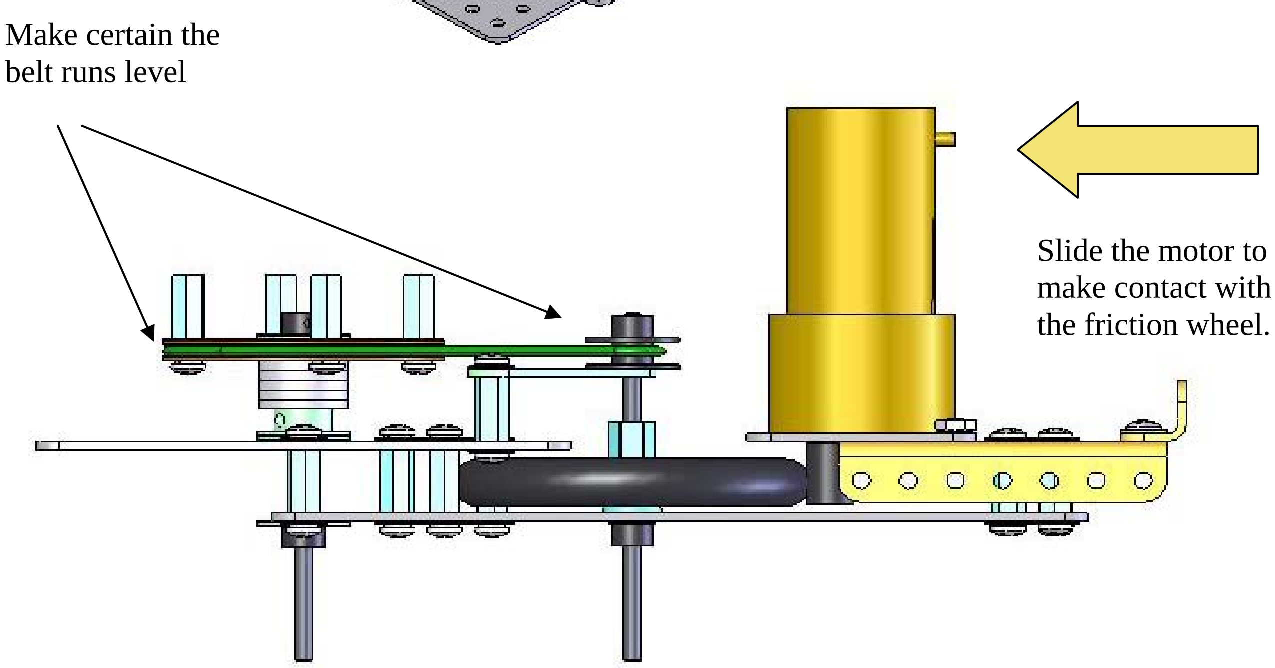




Attach the green polycord belt. Wrap a length of the polycord belt around both the small pulley on top of the friction wheel axle and the 3” hex wheel on the azimuth assembly to find the approximate polycord length. We will call this the wrapped length. Since the polycord stretches, it is necessary to cut the polycord 10% shorter than the wrap length. To do this measure the wrap length of poly cord, and subtract 10% of that length. Example: The wrap length equals 15-1/2”. Convert that length to a decimal, 15.5”. 10% of 15.5” is 1.55”. Round this off to 1 decimal place, 1.5”. and subtract 1.5” from the wrap length. 15.5” – 1.5” = 14”. Cut the polycord to 14” and “Weld” the ends together. Instruction on how to weld the polycord can be found at the end of this document.

Engage the motor and drive wheel by loosening the (4) four motor mount screws and pushing the motor shaft gently against the drive wheel. Connect the motor leads to a battery and test the operation of the system. Make the necessary adjustments.





105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Altitude Positioning Module

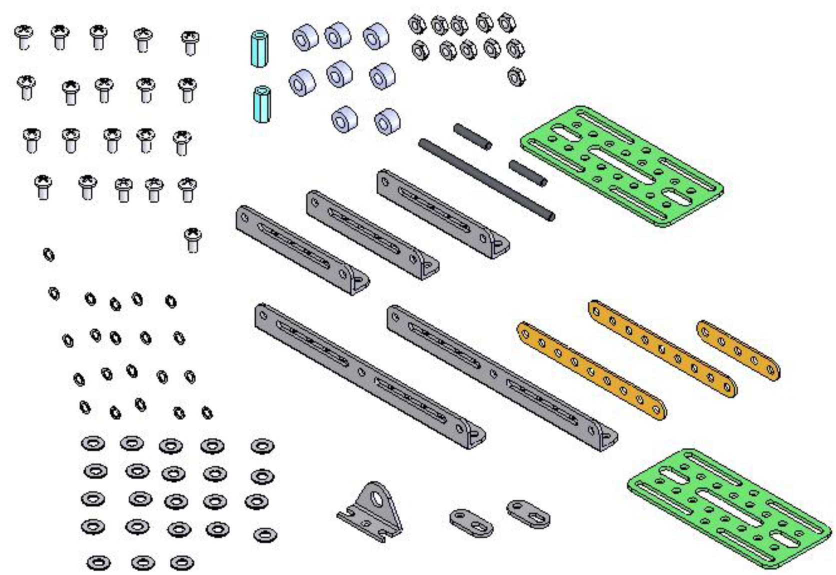
The altitude positioning assembly is operated by a basic 5 component pneumatic system that includes an; air **reservoir** *(energy storage)*, a **3/2 manual valve**, self relieving **regulator**, **3/2 solenoid valve**, and a linear **pneumatic actuator**. Refer to either the pneumatic assembly instruction included in this document.

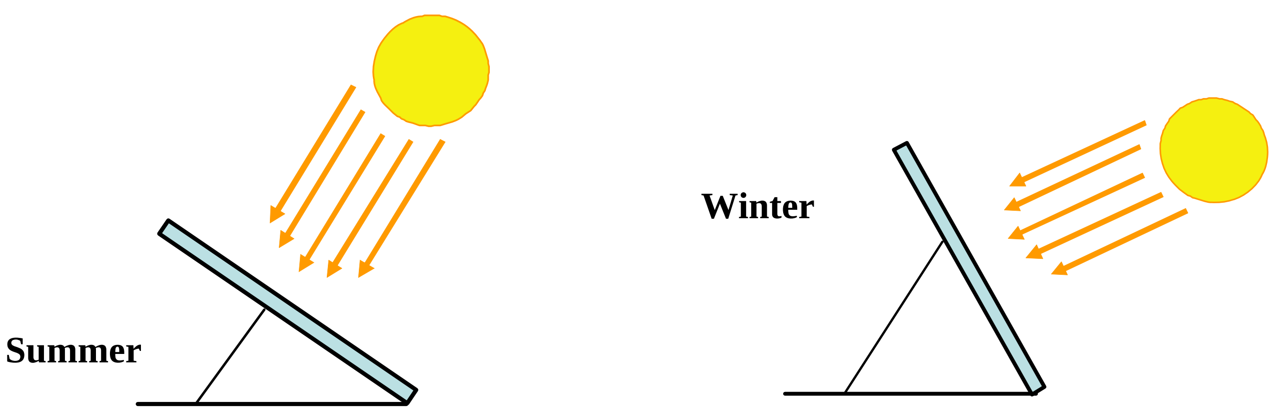
The pneumatic system provides 2 altitude positions, winter and summer. Because the sun is lower on the horizon in the winter months, the attitude angle is greater with respect to the ground plane. In the summer months the sun transits higher across the sky and the altitude angle is less with respect to the ground plane.

Construct the Altitude Positioning Assembly

<u>Necessary Components</u>

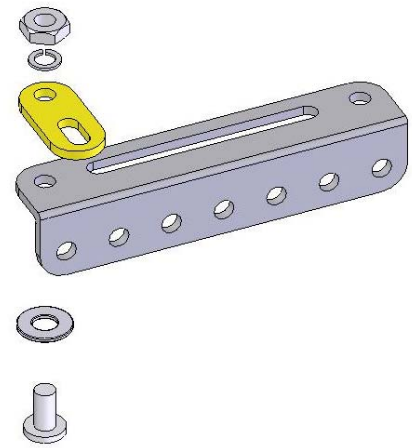
Qty.	Description
23	#10 Flat Washers
21	#10 Lock Washers
11	#10-24 Hex Nuts
21	#10-24 x 1/2" Machine Screws
2	#10-24 Coupling Nuts
8	3/16" Shaft Collars
2	3/16" dia. x 1-1/2" axle
1	3/16" dia. x 4" axle
3	7 Hole Angles
2	13 Hole Angles
2	Bearing Plates
2	9 Hole Flat Bar
1	5 Hole Flat Bar
2	180 Degree Fish Plates
1	Pneumatic Bracket
1	Linear Actuator (Pneumatic)





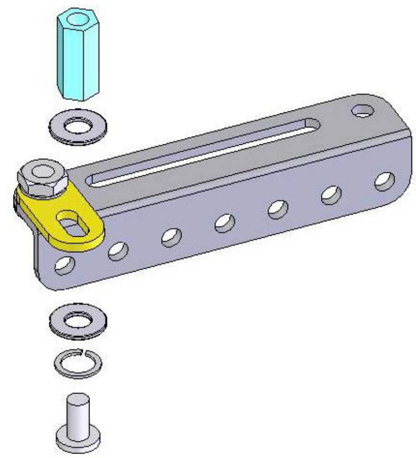
Procedure

Step One: Make the Altitude Positioning Assembly Base
The base sits atop the 4 coupling nuts on the 3" hex wheel.



1. Lay out the following components from the list on the preceding page.

Qty	Description
8	#10-24 x ½" ph machine screws
9	#10 Flat washers
6	#10-24 Hex nuts
2	#10-24 Coupling nuts
8	#10 Lock washers
2	7 Hole angles
2	9 Hole flat bar
2	180 degree fish plates

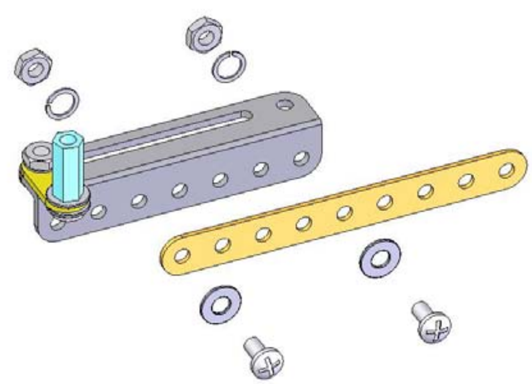


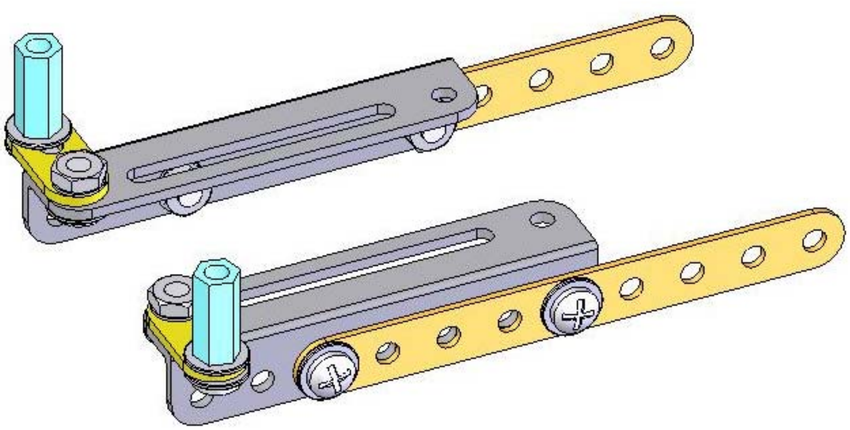
2. Fasten the 180 degree fish plate to the 9 hole angle using a machine screw, flat washer, lock washer and hex nut in the order shown on the right.

3. Fasten the #10-24 coupling nut to the fish plate. Use a flat washer on both sides of the fish plate. Always include a lock washer.

4. Attach the 9 hole flat bar to the 7 hole angle as shown below and right. Be certain to count the holes and position the flat bar exactly as shown. The finished assembly is shown below.

5. Repeat steps 1 through 4 and assemble the other side of the base. Remember that these two sides are mirror opposites, assemble them accordingly.

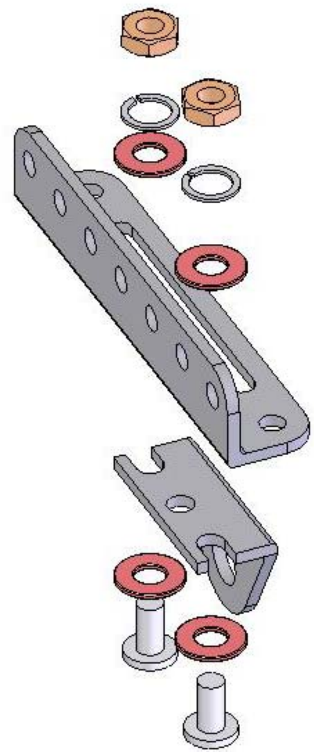




Step Two: Assemble the Pneumatic Actuator Mount
The mount sits atop the 2 #10-24 coupling nuts on the base (see bottom of preceding page.

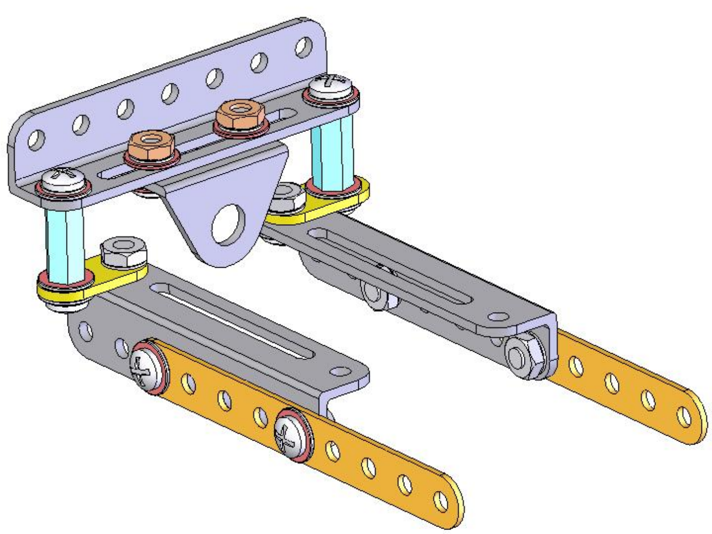
1 Lay out the following components.

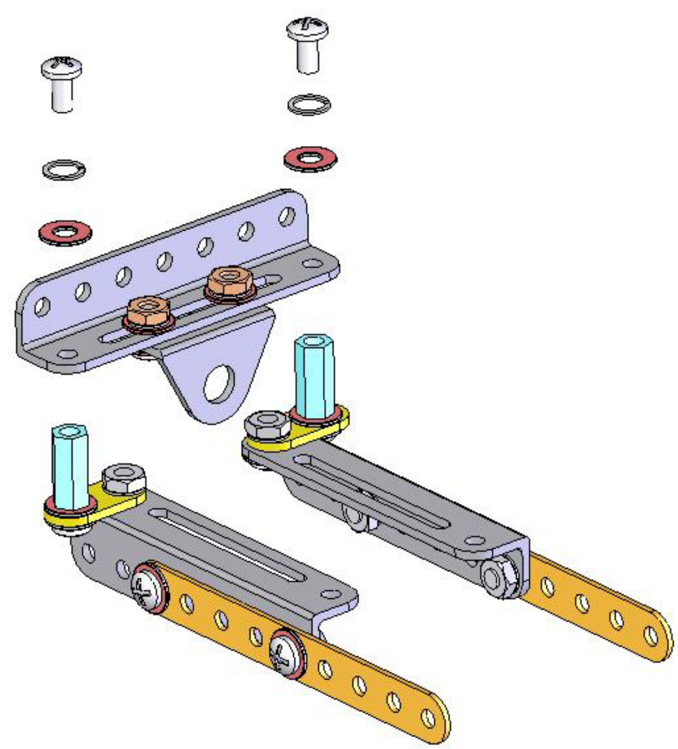
Qty	Description
2	#10-24 x ½” ph machine screws
4	#10 Flat washers
2	#10-24 Hex nuts
2	#10 Lock washers
1	7 Hole angles
1	Pneumatic bracket



2. Fasten the pneumatic bracket to the center of the 7 hole angle. Use flat washers on both sides of the 7 hole angle. Do not fully tighten the fasteners at this time. It may be necessary to slide them side to side in order to position the pneumatic actuator.

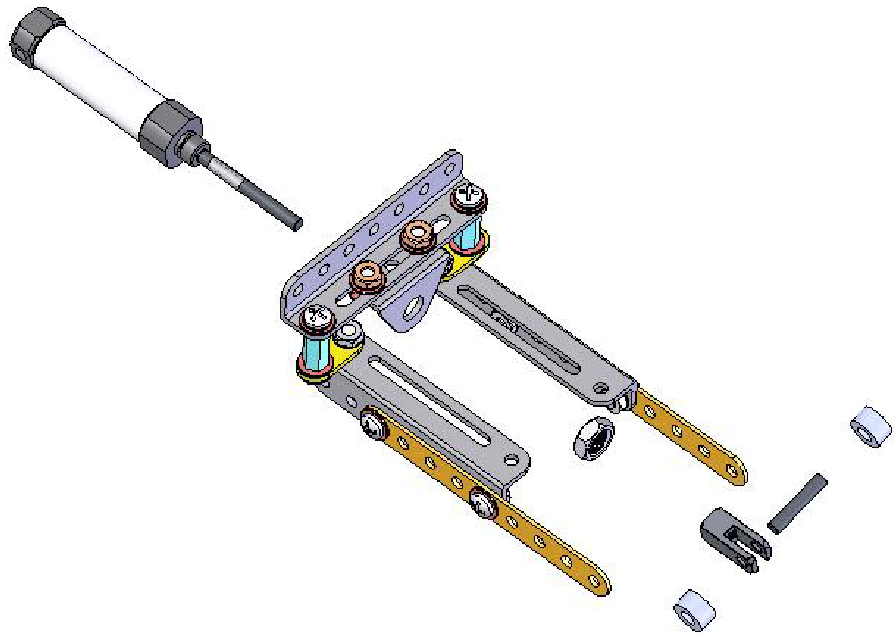
3. Fasten the completed pneumatic bracket to the base as shown below.

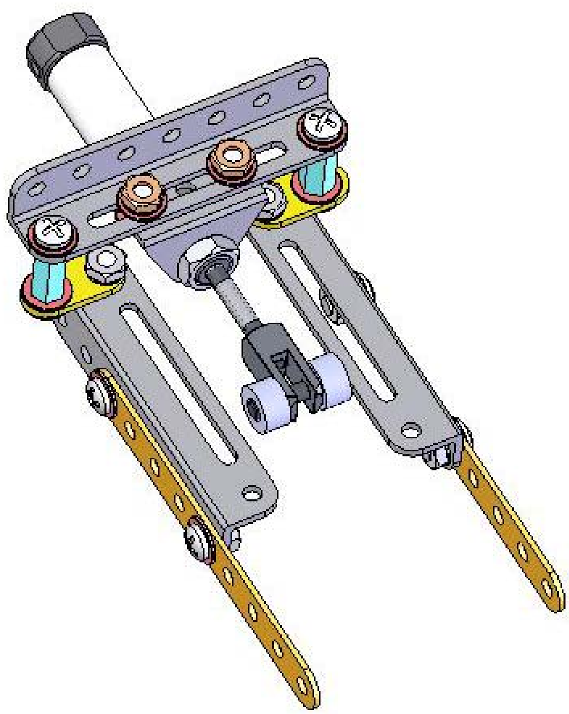




105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Step Three : Attach the Pneumatic Actuator to the Mounting Bracket and Base





Necessary Components

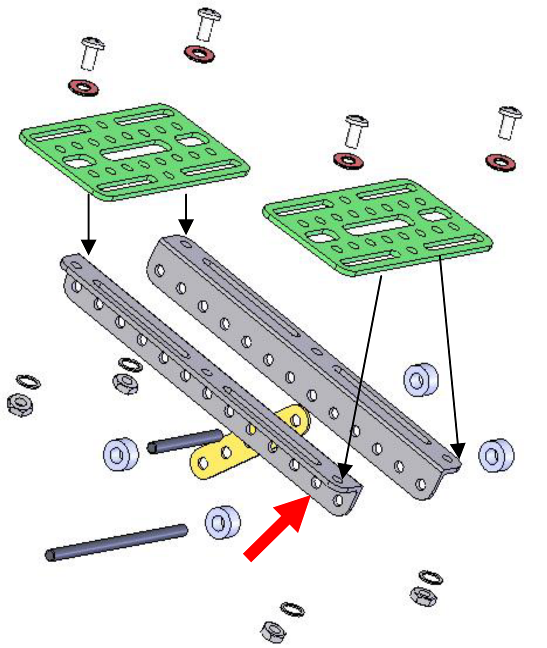
1	Pneumatic actuator
1	3/8" Hex nut
1	Clevis
1	3/16" Diameter x 1" axle
2	3/16" Shaft Collars

Fasten the pneumatic actuator to the pneumatic bracket using the 3/8" hex nut.
Thread the clevis onto the rod end and assemble the axle and shaft collars. Note: do not fully tighten the shaft collar set screws at this time. It will be necessary to remove the axle at a later time in order to mount the altitude module. The completed pneumatic actuator assembly is shown on the left.

Step Four : Assemble the Solar Panel Mounting Bracket

The solar panel mounting bracket supports the solar panel.

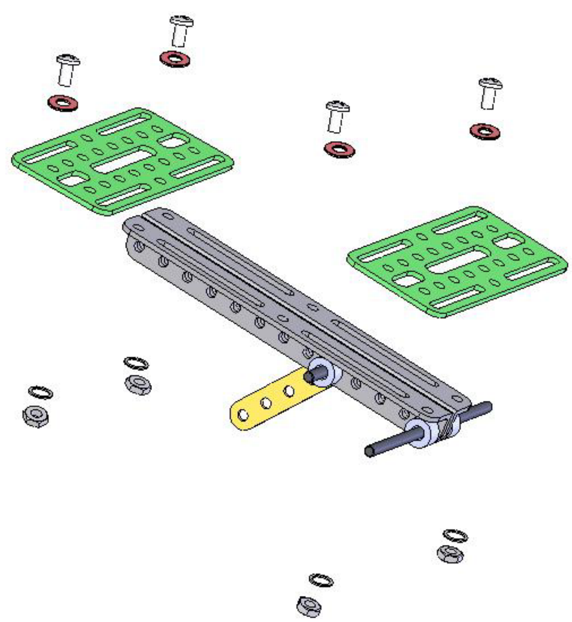
Necessary Components



Qty	Description
4	#10-24 x ½" ph machine screws
4	#10 Flat washers
4	#10-24 Hex nuts
4	#10 Lock washers
2	13 Hole angles
2	Shaft Plates (Green)
1	5 Hole flat bar (yellow)
1	3/16" Diameter x 1" axle
1	3/16" Diameter x 4" axle
4	3/16" Shaft collars

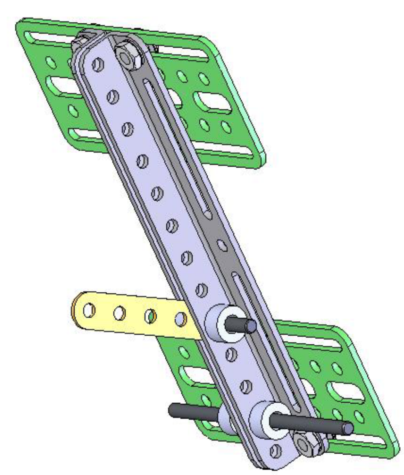
Pass the 4" axle through the second hole up from the end of the 13 hole angle. Refer to the larger red arrow in the illustration on the left.

Sandwich the 5 hole flat bar (yellow) in between the two angles and use the 1" axle and two shaft collars to fasten the flat bar in place as shown on the left.



Fasten the shaft plates (Green) to the 13 hole angles. Mount the shaft plates to the holes on each end of the 13 hole angles. Refer to the arrows in the illustration on the left.

The completed solar panel mounting bracket is shown on the right.



Rear (isometric) **view of the completed assembly**

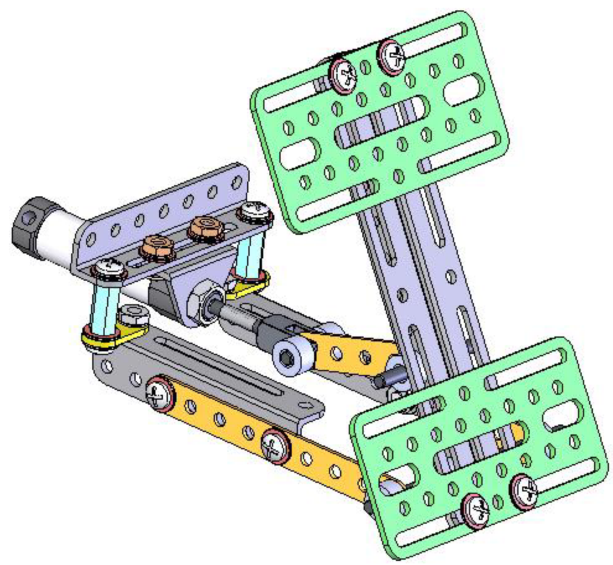
105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

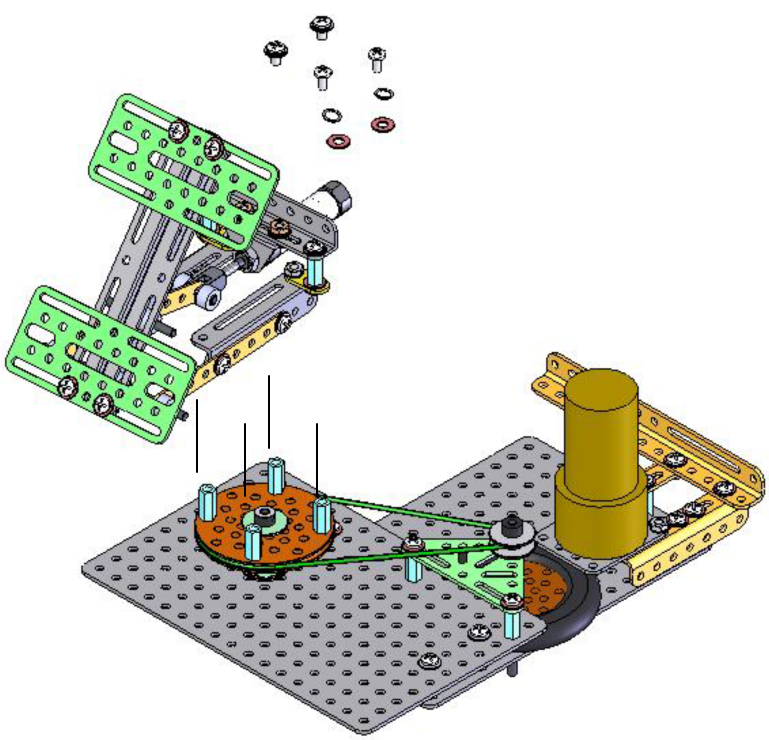
Step Five: Integrate the Pneumatic Actuator Mount and the Solar Panel Mounting Bracket Assemblies

The assembled altitude positioning module is shown on the right. This unit allows for two angle positions *(measured from the horizontal).*

One angle can be set *low* for summer sun positions, and one angle can be set *high* to accommodate winter angle positions.

The two angles can be adjusted by positioning the 5 hole flat plate linkage either higher or lower on the solar panel mounting bracket.



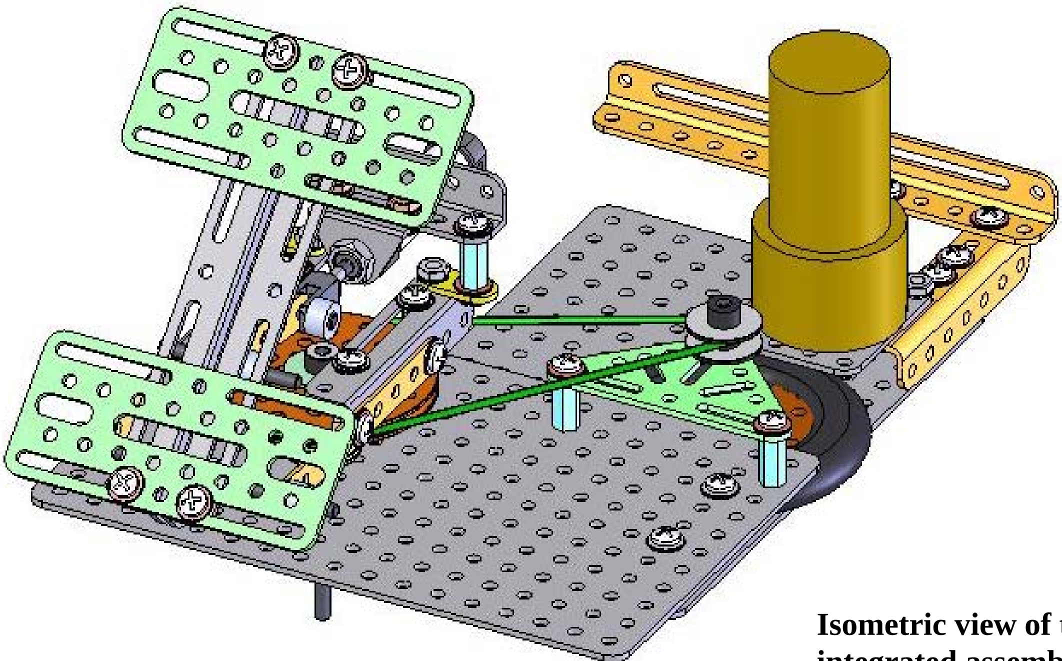


Necessary Components

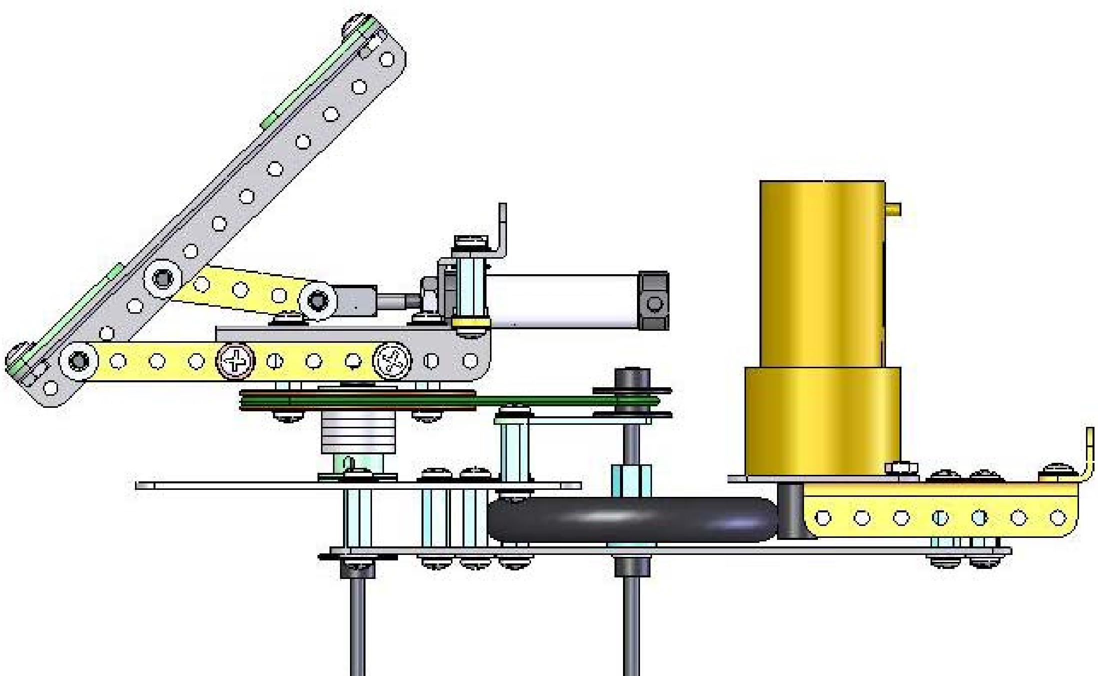
Qty	Description
4	#10-24 x ½" ph machine screws
4	#10 Flat washers
4	#10 Lock washers

Fasten the solar panel mounting bracket to the *10-24 coupling nuts on the 3" hex wheel (orange). This is done by passing the ph machine screws through the slots in the 7 hole angles. Position the washers under the screw heads and above the slots.

The complete assembly is shown on the following page.



Isometric view of the integrated assembly



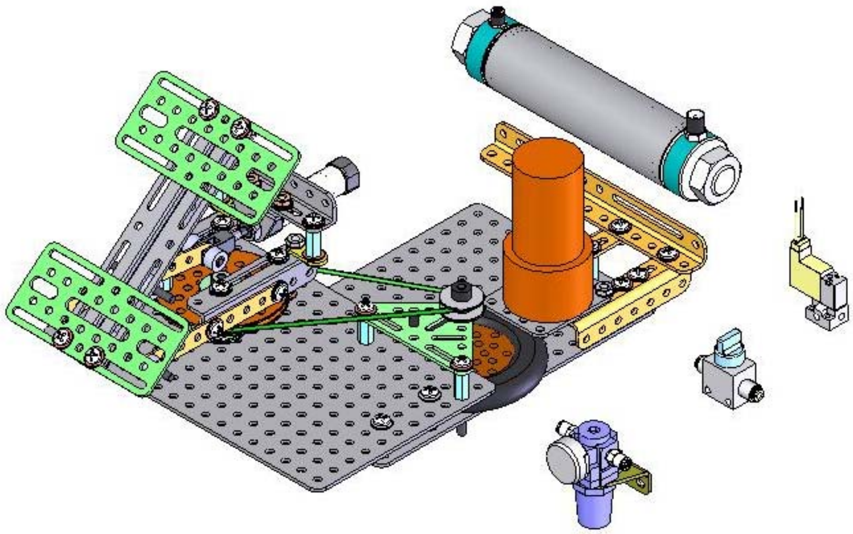
Right side view of the integrated assembly

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Step Six: Install the Pneumatic Components

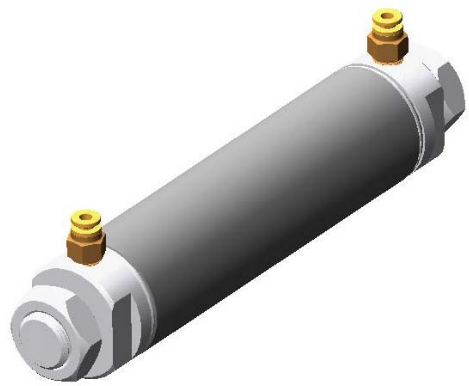
Necessary Components

Qty	Description
1	Regulator
1	Pneumatic Reservoir
1	3/2 Manual Valve
1	3/2 Solenoid Valve
24"	4mm tubing (Not Shown)



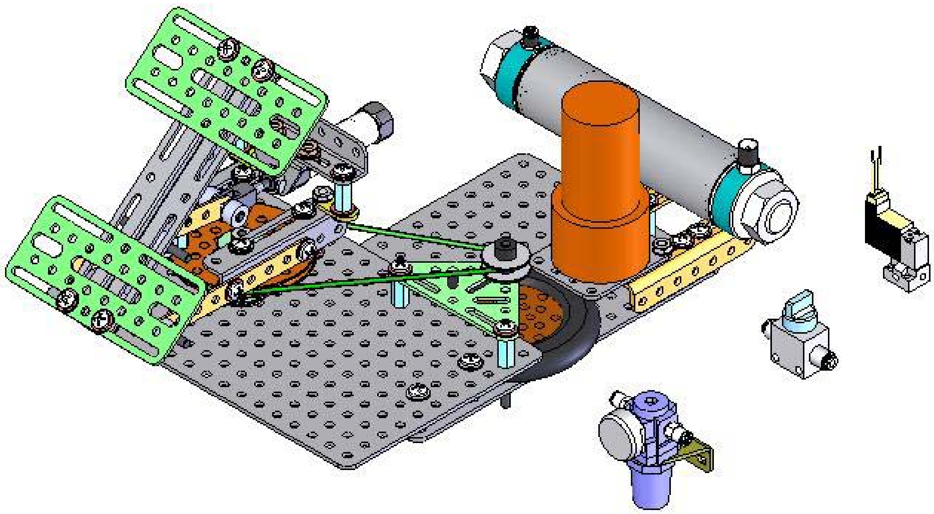
Note: Take the time to become familiar with the purpose and function of each of the pneumatic components. Assemble and operate a "Bench top" working pneumatic system before attempting to integrate the components into a working solar tracker.

Mounting the Pneumatic Components



The Reservoir

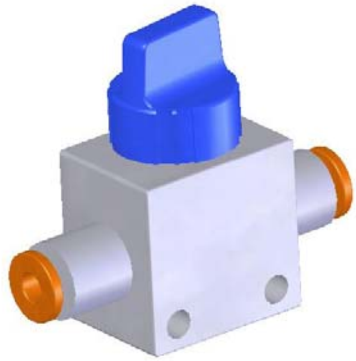
The reservoir is the equivalent of a battery. The energy used by the cylinder to operate the altitude positioning module, is stored as compressed air in the reservoir. You can "Feel" this "captured" energy when you pressurize the cylinder using a bicycle pump.



Mount the reservoir lengthwise along the 13 hole angle plate that runs along the back of the solar tracker assembly. Secure it in place using zip ties, elastic or other readily available means. See the illustration on the left.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

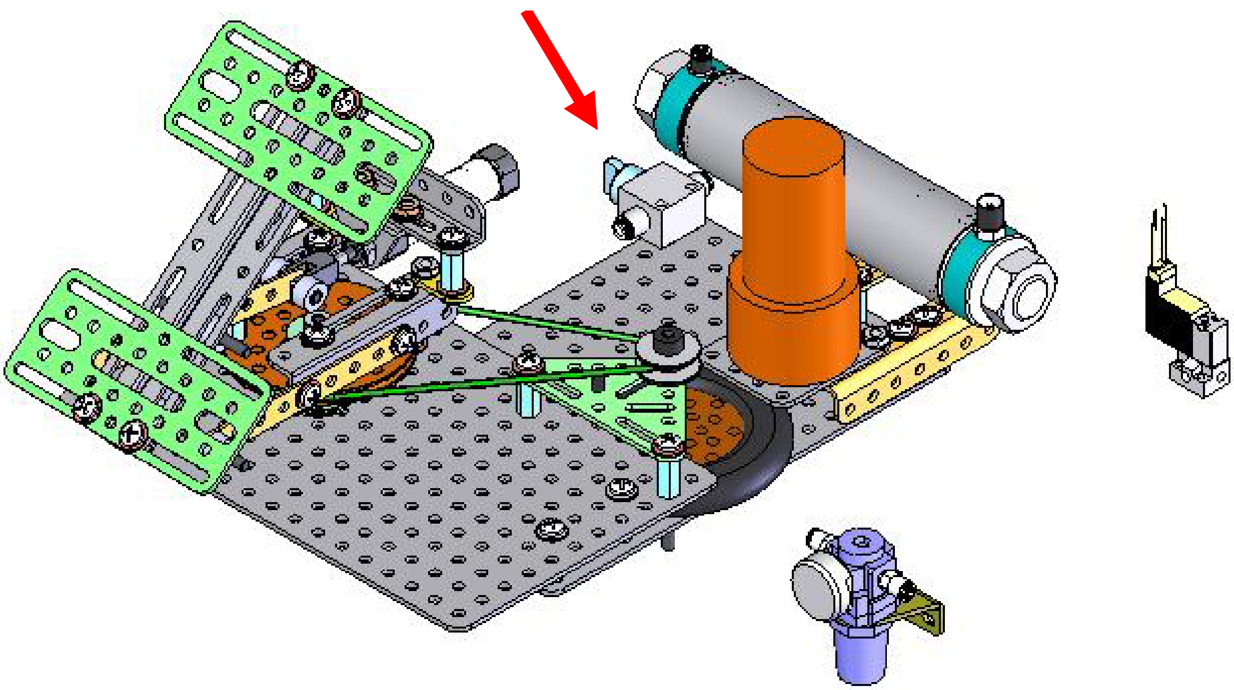
The 3/2 Manual Valve (Relieving)



The 3/2 manual valve is attached to the reservoir using with a length of 4mm tubing. Note the air flow direction arrow embossed on the valve body. These arrows show the direction of air through the valve. Attaching the valve in reverse will cause the reservoir to discharge.

This valve acts like a pneumatic "on, off" switch with 2 different positions and 3 ports or holes, for the air to pass through. When the valve is turned on, the system is pressurized and ready to operate. When the valve is off, the compressed air contained in the pneumatic components downstream of the 3/2 manual valve are depressurized. (The system is vented through the valve to atmosphere) This ensures the downstream components are safe and not energized. Mount the 3/2 manual valve using 2 #4-40 x 1" machine screws, nuts and washers. Place it close to the reservoir and in a position where the valve can be conveniently operated.

The 3/2 Manual Valve (Relieving)
Located near the reservoir. Remember to check the air flow direction arrow embossed on the valve body.



The Self Relieving Regulator and Gauge

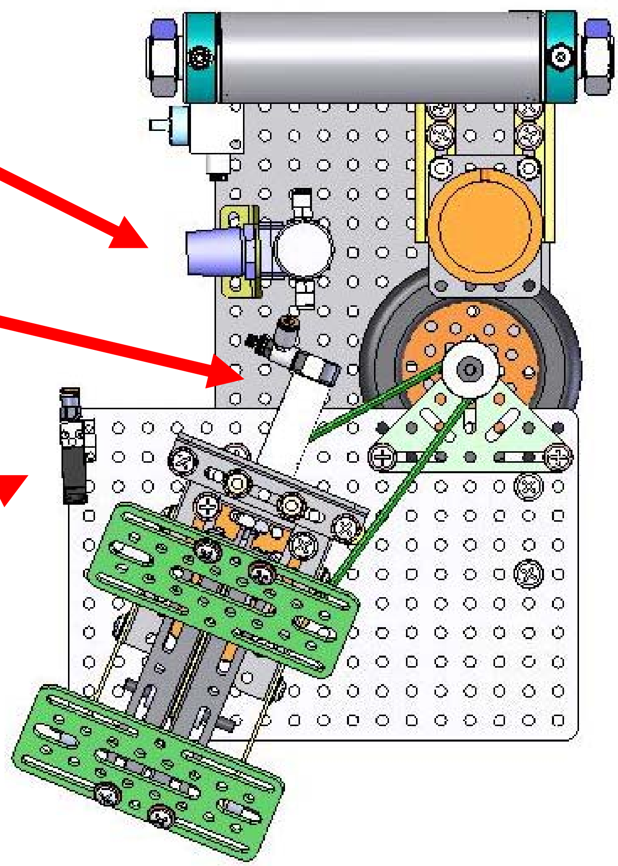
The regulator is connected to the 3/2 manual valve. The regulator is used to maintain a constant working pressure in the system. It is set to maintain a fixed pressure and will continue to do so until the pressure in the reservoir drops below the working pressure in the system. The reservoir will provide a finite amount of operating cycles with a given reservoir pressure. The number of possible cycles for a given reservoir pressure can be approximately determined using Boyle's law.

The regulator serves as an important safety device. If the pressure in the system increases beyond the regulator set point, the regulator will vent the excess pressure. **It is unwise to operate a pneumatic system without using a self relieving regulator with a gauge.**

Mount the regulator near the 3/2 valve using two (2) #6-32 x ¾" machine screws nuts and washers.

The self relieving regulator with gauge mounted near the 3/2 manual valve. Remember; the correct air flow direction through the regulator is marked with an arrow embossed on the back of the regulator casing.

The single acting pneumatic cylinder or "Linear actuator" with a flow valve attached.

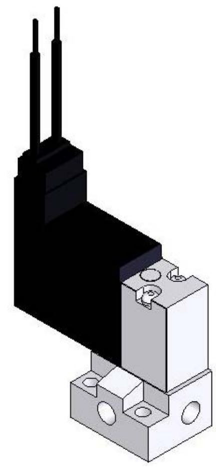


The 3/2 Solenoid Valve. The air inlet is marked with an embossed letter "P" or the number "1".

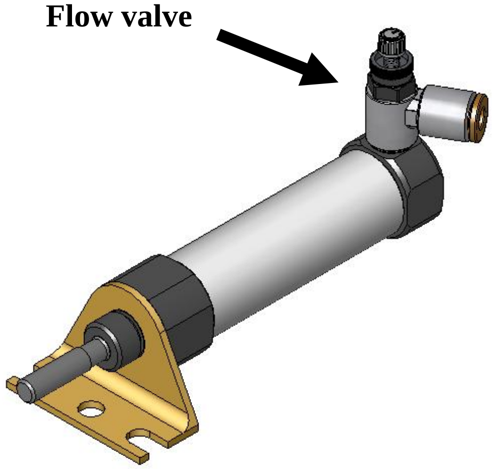
Top View of the Solar Tracker Module

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

The 3/2 Solenoid Valve
The 3/2 solenoid works like the 3/2 manual valve; only it is operated electrically. This convenient feature allows the valve to be operated remotely or autonomously.



Flow valve



The Single Acting Pneumatic Cylinder

This is the "Work horse" of the pneumatic system. If the reservoir is analogous to a battery, and the 3/2 valves are like switches, then the pneumatic cylinder can be though of as being like the motor in this system. The pneumatic cylinder is also referred to as a linear actuator since it applies force in a straight line or linear direction.

The term single acting is used to describe a linear actuator that applies a working force in one direction. A spring is used to return the cylinder rod to the starting position. A double acting cylinder can be used to supply force in two directions. A double acting cylinder requires an different solenoid valve.

The pneumatic cylinder sold with the GEARS kit also includes a flow valve. The flow valve is also referred to as a speed valve. A needle valve, turned by a thumb screw is used to regulate the flow of air into the cylinder. The rate of air flowing into the cylinder determines the speed of the rod and piston. The flow valve can be set so that the cylinder rod extends very slowly. The thumb screw can be set using a locking collar.

Plumbing and Testing the Pneumatic System

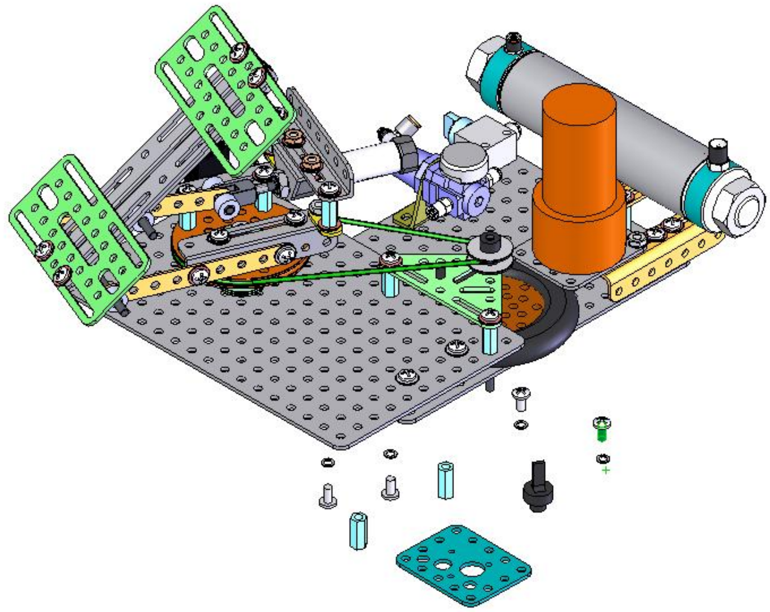
After the pneumatic components have been installed complete the pneumatic (series) circuit by connecting them with the 4mm tubing. The order of connection is as follow: Reservoir > 3/2 manual valve > self relieving regulator > 3/2 solenoid valve > pneumatic cylinder with speed valve. Remember that each of these pneumatic components has a input and an output port, and if they are reversed, the system will not work.

When the system components have been properly connected and checked, pressurize the system with a bicycle pump. A bicycle pump is recommended for two important reasons
1.) It allows slow and controlled pressurization so that any leaks or connection errors can be discovered before the pressure rises too high.
2.) The person pressurizing the system can "feel" the work or energy being put into the system.

Step Seven: Assemble and Attach the Single Pole Single Throw Switch

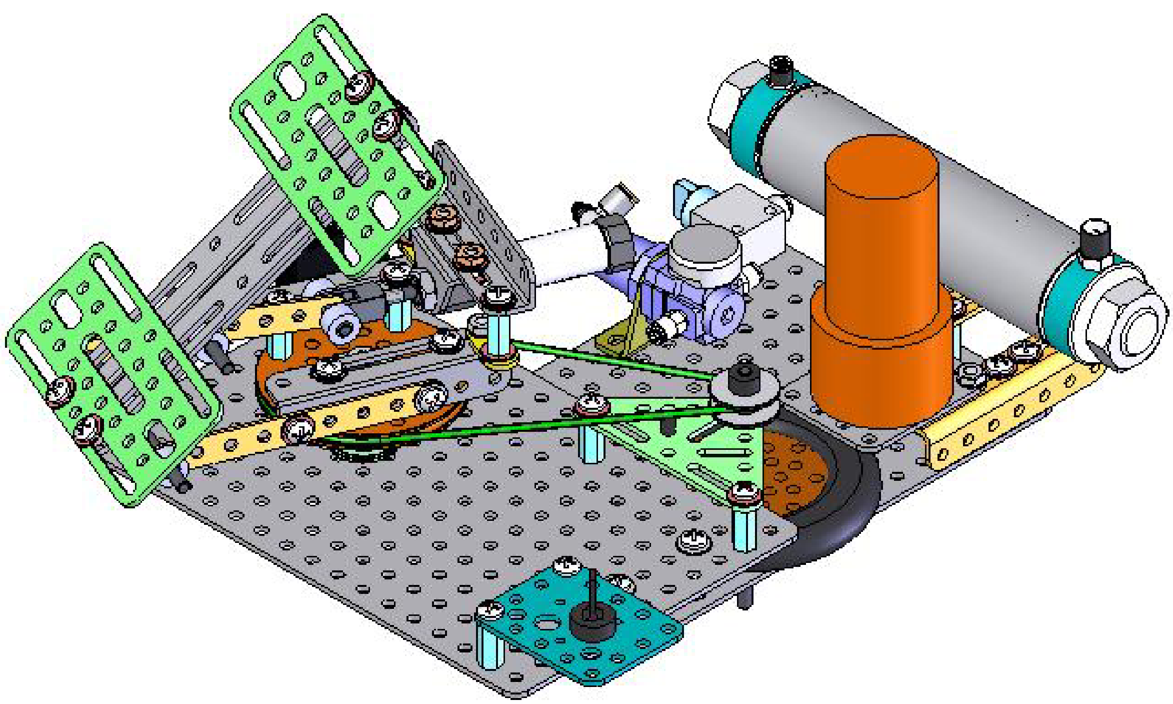
Necessary Components

Qty	Description
1	SPST Toggle switch
4	#10-24 x 3/8"ph machine screws.
4	#10 lock washers
2	#10-24 Coupling nuts
1	Switch plate



Procedure

Fasten the switch plate to the solar tracker module in a position that will not interfere with the (azimuth) rotation of the solar tracker.



Adding Controls

The solar tracker requires two “Channels” of control; One channel operates the motor and controls the azimuth axis, the second channel operates the pneumatic system and controls the altitude axis. Azimuth control requires continuous positioning through approximately 180 degrees of rotation (Actual rotation angles depend on latitude and time of year.The altitude control requires only two positions optimized for either summer/winter or morning/noon/evening sun angles.

Radio control

For testing purposes it is possible to control both channels using an RC radio. For real world solar tracking experiments, it is necessary to automate the operation of the solar tracker. This is best accomplished using a microprocessor.

Microprocessor Control
Any number of control strategies can be developed. The simplest solution would be a “Timed” operation in which the control system is positioned each day facing easterly, and the control system is turned on and allowed to slowly follow the sun across the sky. This strategy may work, but it would provide limited applicability, and it would be prone to tracking errors caused by difficulties in synchronizing the tracker rotation to precisely follow the sun.

A better strategy would be to provide feedback through a system of sensors programmed to search for an optimum position for the tracker. There are several ways of doing this, but they can be readily accomplished using an number of available microprocessor systems. We recommend either of these two great educational microprocessor systems:

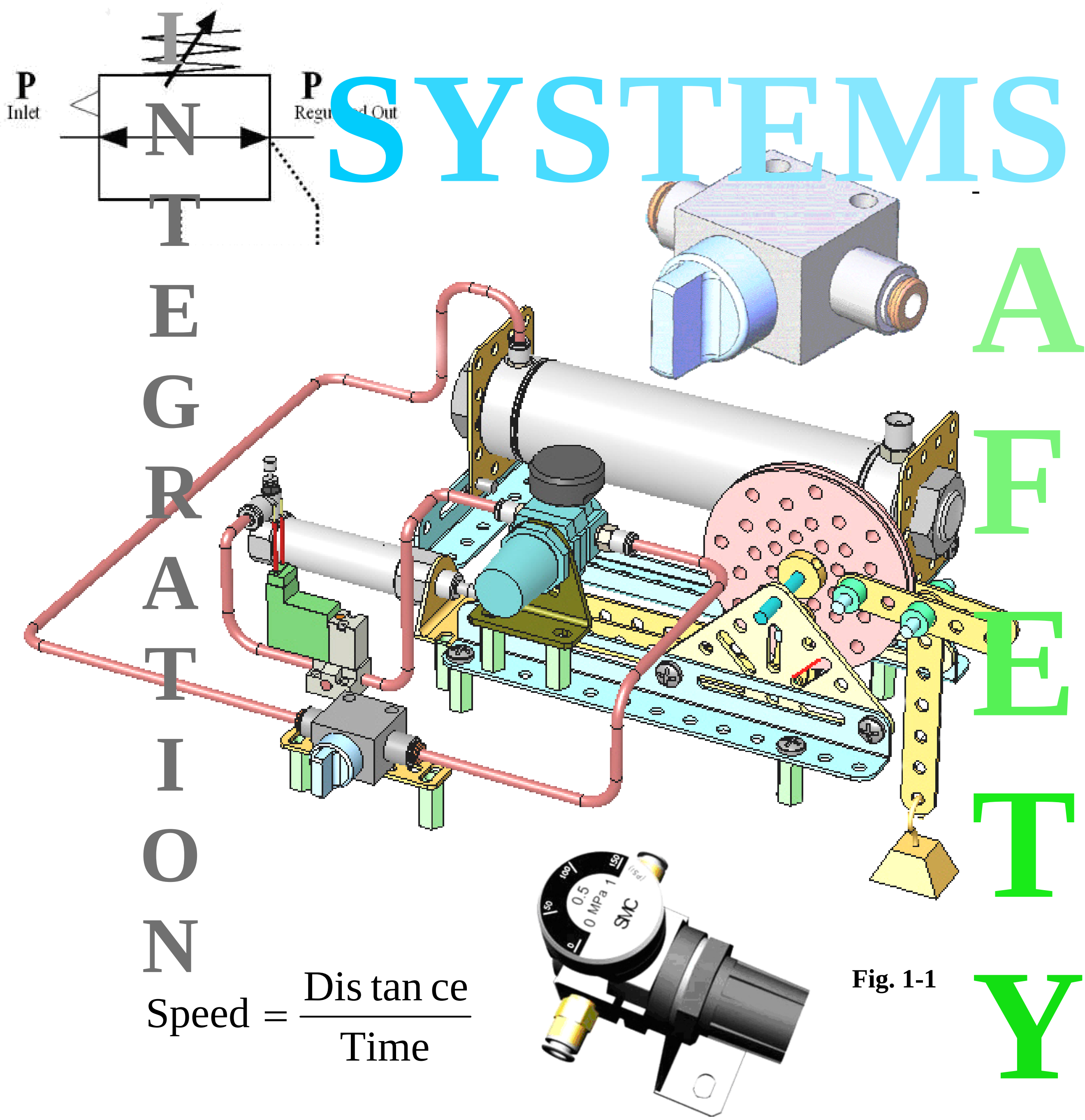
Parallax Basic Stamps and Propeller

The BASIC Stamp module is a microcontroller developed by Parallax, Inc. which is easily programmed using a form of the BASIC programming language called PBASIC. It is called a “Stamp” simply because it is close to the size of an average postage stamp, except for the BS2p40 which is much longer due to it's additional I/O pins.

The Propeller Chip makes it easy to rapidly develop embedded applications. Its eight processors (cogs) can operate simultaneously, either independently or cooperatively, sharing common resources through a central hub. The developer has full control over how and when each cog is employed; there is no compiler-driven or operating system-driven splitting of tasks among multiple cogs. A shared system clock keeps each cog on the same time reference, allowing for true deterministic timing and synchronization. Two programming languages are available: the easy-to-learn high-level Spin, and Propeller Assembly which can execute at up to 160 MIPS (20 MIPS per cog).

Machine Science
Machine Science’s C based microprocessor kits and resources are used in a wide range of educational programs—running after-school programs for middle school students, prototyping commercial products in undergraduate engineering labs, fielding entries in open-architecture robotics competitions, and developing innovative technology projects at home, to name just a few. To learn more about these and other applications, please click on the Machine Science header at the top of this paragraph.

Section Two: Configure and Operate Pneumatic Components



$$\text{Speed} = \frac{\text{Distance}}{\text{Time}}$$

Fig. 1-1

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Basic Pneumatic System Components

Regulator (Self relieving)

<u>Regulators control circuit pressure or force.</u> Pressure is a measure of force acting over a specific area P = force/area. These devices are fitted with mechanical components that react to changes in the downstream air pressure. The regulator attempts to automatically maintain a constant (preset) pressure within a pneumatic circuit as long as the supply (reservoir) pressure is greater than the required circuit pressure. The reading on the regulator-mounted gauge indicates the regulated or circuit pressure

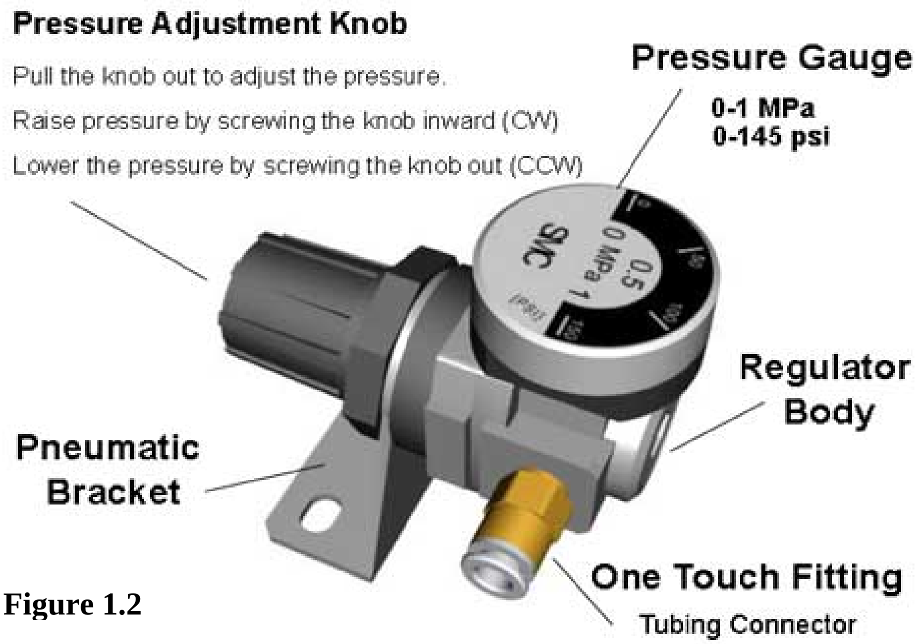


Figure 1.2

Regulator Controls Pressure or Force

Note: Always use a regulator and a pressure gauge to monitor and control pneumatic system pressure. Every pneumatic system should have a pressure relief valve to prevent over pressure conditions. A self relieving regulator is designed to vent overpressure conditions on the downstream side of the pneumatic circuit. The regulator used in the GEARS-IDS™ kit has a self relieving feature.

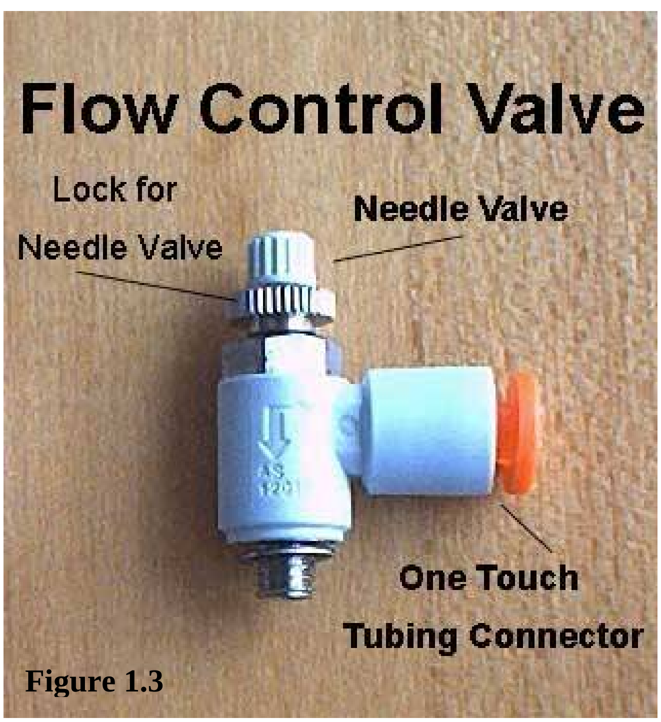


Figure 1.3

Speed or Flow Valves

<u>Flow valves control the speed of air flow into or out of a pneumatic circuit or component.</u> Flow is a measure of the volume of air moving through the circuit or component over a period of time (Flow = volume/time). Flow control is adjusted using the needle valve. Screwing the needle valve outward increases the flow rate, the higher the flow rate, the faster the component will operate.

Note: Air enters and leaves the single acting cylinder through the flow control valve. Airflow is regulated in one direction only. The free flowing air direction is shown using a large arrow embossed on the valve. The regulated air flow direction is shown with a small arrow.Airflow control is best accomplished by regulating the flow of air <u>out of a circuit or component</u>. Note: Controlling air flow out of the cylinder is the preferred choice for accurate and smooth control of slower moving actuators. "When in doubt, regulate out!".

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Single Acting Pneumatic Cylinder or Linear Actuator

These devices are used to apply straight line (linear) pushing or pulling forces. Linear actuators are available in thousands of different configurations. These cylinders are fitted with pistons of various diameters and strokes of various lengths. They are most commonly specified as single acting (powered in one direction) or double acting (powered in both directions). Single acting spring return cylinders are more economical with respect to air consumption. The pneumatic cylinder supplied in the GEARS-IDS Invention and Design System is a single acting, spring return cylinder. (see Figure 1.4 and 1.5)

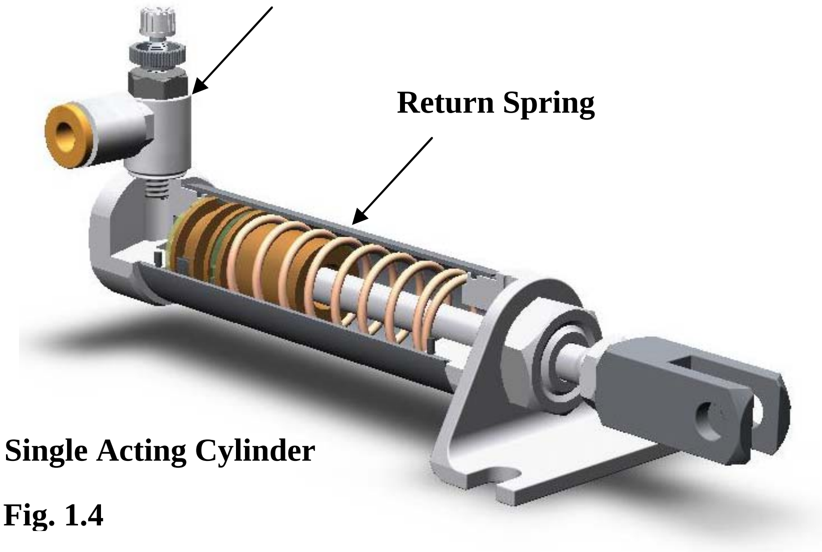


Fig. 1.4

The pneumatic cylinder used in the GEARS-IDS™ kit has a bore (Interior diameter) of 16 millimeters or 0.629". Since 5/8" = 0.625, this cylinder can also be referred to as a 5/8" bore cylinder for computational purposes. When pressure is applied to the piston, the cylinder rod extends outward 25.4 millimeters or 1.0". Important values to consider when designing or evaluating pneumatic system performance are the surface area of the piston and the interior volume of the cylinder when the piston rod is fully extended. The interior volume of the cylinder is determined by calculating the surface area of the piston and multiplying the area of the piston by the length of the stroke.

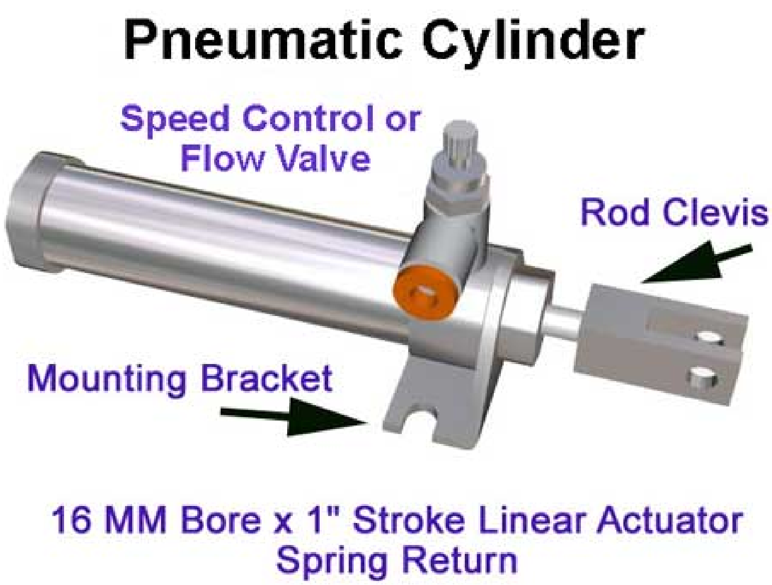


Fig. 1.5

Determine the Surface area of the piston and the interior volume of the cylinder using the following formula:

$$Area = \pi * R^2$$

$$Volume = \pi * R^2 * Length_{cylinder}$$

Notebook Exercise: Draw a sketch of the GEARS-IDS™ cylinder. Include all the dimensions and calculations necessary to correctly determine the interior volume of the cylinder.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

3-2 NC *(Normally closed)* Solenoid Valve

Solenoid valves are electrically operated valves that control the direction and flow of pressurized air to and from pneumatic actuators or circuits. Solenoid valves can be either mono-stable, *(they spring return to a default condition either on or off*) or Bi-stable, *(having no preferred or default condition thus remaining where it was last positioned either on or off*) Pneumatic valves can be operated by hand, *(mechanical)* electrically *(solenoid)* or air *(piloted)* operated. The GEARS-IDS™ kit includes a 3 port, 2 position electrically operated solenoid valve.

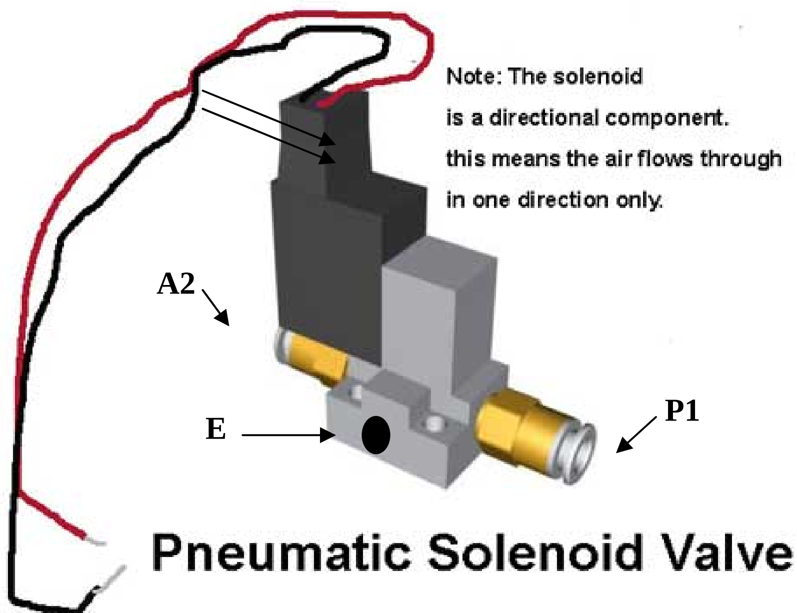


Fig. 1.6

The GEARS-IDS™ 3-2 pneumatic solenoid valve is described using 2 numbers. Example; The solenoid valve included in the GEARS-IDS™ kit *(pictured in fig 1.6)* is referred to as a 3-2 solenoid valve. This means the valve has 3 ports *(P1, A2 and E)* and 2 possible conditions *(Passing or not passing)* and it is electrically operated *(Solenoid).*

Ports and Positions of a 3-2 Valve

The first number 3, refers to the number of ports or holes through which air moves into or out of the valve and the 2 refers to the number of valve positions or conditions.

Examine the valve closely. You will find 3 holes or ports in the base of the valve body. They are usually labeled as P1, A2 and E. The port labeled P1 is the pressure or inlet port. P1 connects to the pressure supply. The A2 port supplies pressurized air from P1 to an actuator or a circuit and in turn, allows air to pass from an actuator or a circuit to the E or exhaust port. The E port is open to the atmosphere.

The 3-2 valve has only 2 possible valve positions or conditions; The valve can either be passing air from P1 to an actuator or circuit through A2 *(the open condition)* or, not passing air from P1 but rather passing Air from A2 to the E (exhaust) port *(the closed condition).*

Position One (Default)

When the solenoid's electrical circuit is not energized (default condition), pressurized air cannot pass from the P1 port, through the valve to the actuator or circuit. The air pathway that exits in this (default) condition, connects the A2 port with the E *(Exhaust)* port and blocks the P1 port. In this condition air can only move from the actuator, through the A2 port to the E (Exhaust) port. The E port provides a means for air to exhaust to the atmosphere.(See figure 1.7)

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Figures 1.7 and 1-8 are pictorial representation of a 3-2 solenoid valve. Figure 1-7a is a schematic or symbolic representations of the same valve.

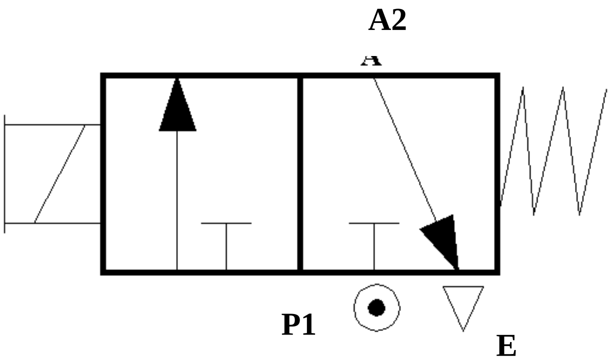
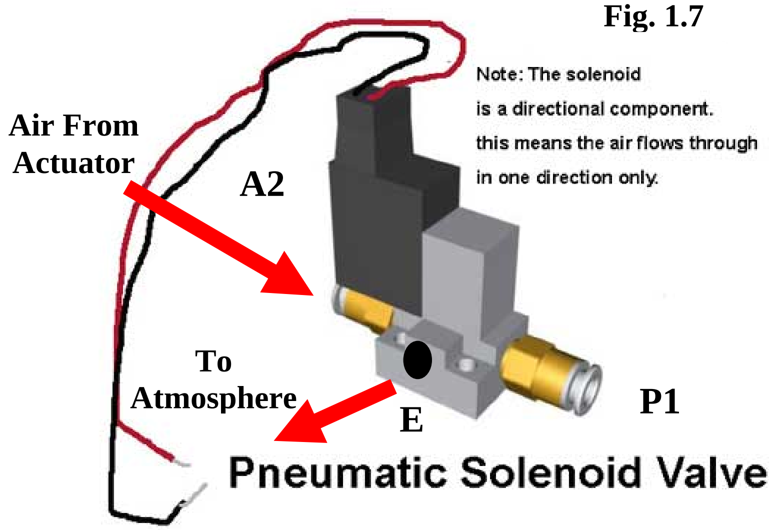


Fig. 1.7



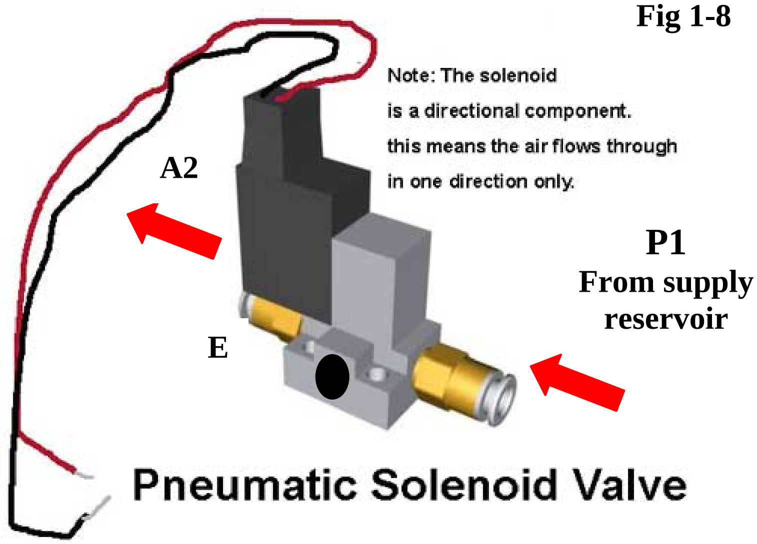
Normallv Closed Position

Position Two (Energized)
(See figure 1.8)

When the solenoid is energized a valve opens creating a pathway or circuit from P1 to A2. In this condition a source of pressurized air can be directed to an actuator or other pneumatic circuit.

When the solenoid is de-energized, the valve reverts to the default (Normally closed) position and the pressurized air in the cylinder is directed out to the atmosphere through the exhaust valve. *(Fig. 1-7)*

Fig 1-8



105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Anatomy of a 3-2 Solenoid Valve Symbol

Valve symbols can be confusing. It may prove helpful to review the pneumatic schematics slide show included with this text.

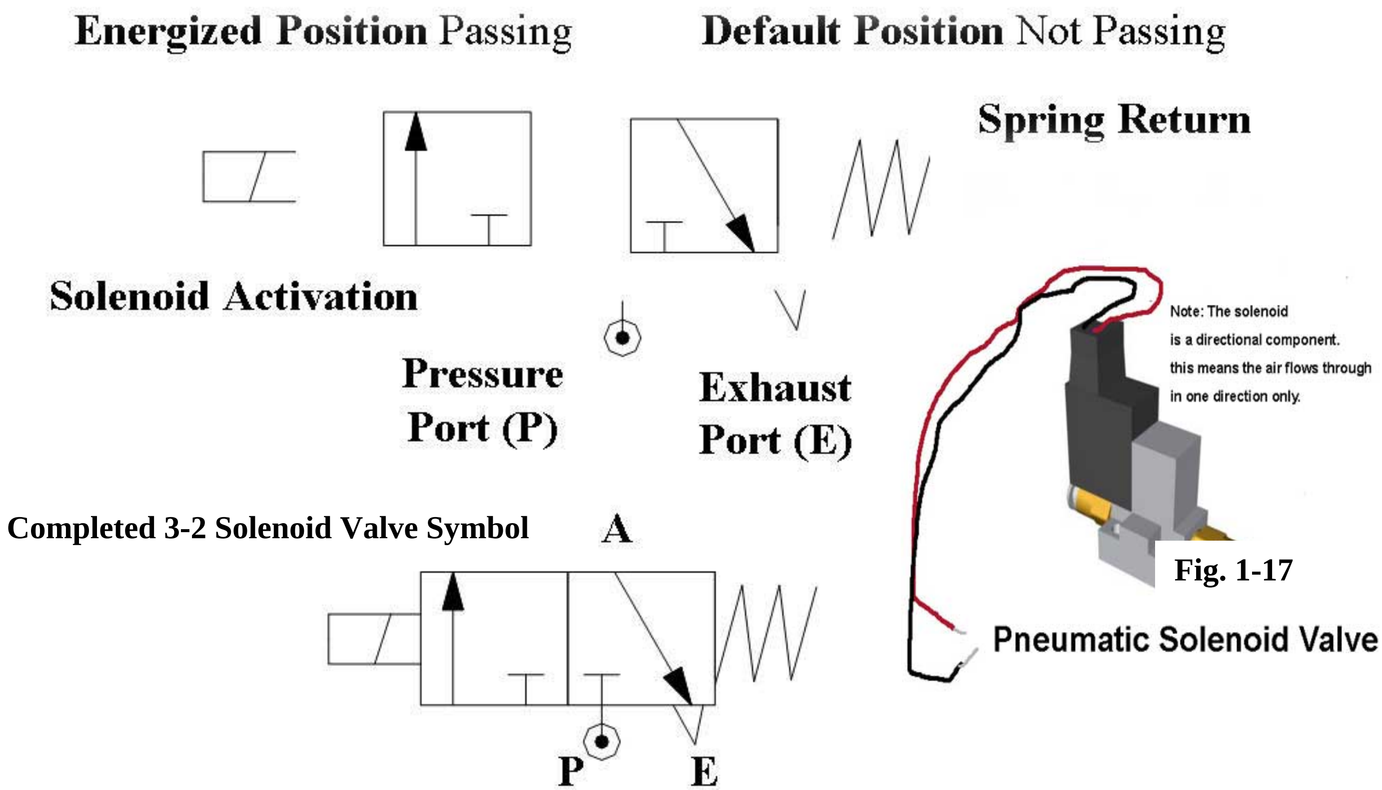


Fig. 1-17

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Air Reservoir

Note: Significant amounts of energy can be stored in pressurized air containers. For this reason you should always wear safety glasses when working with pressurized air systems. In order to prevent over pressurization, use ONLY bicycle pumps to pressurize the air storage containers used with the GEARS-IDS™ pneumatic components. Do not exceed 100 psi.

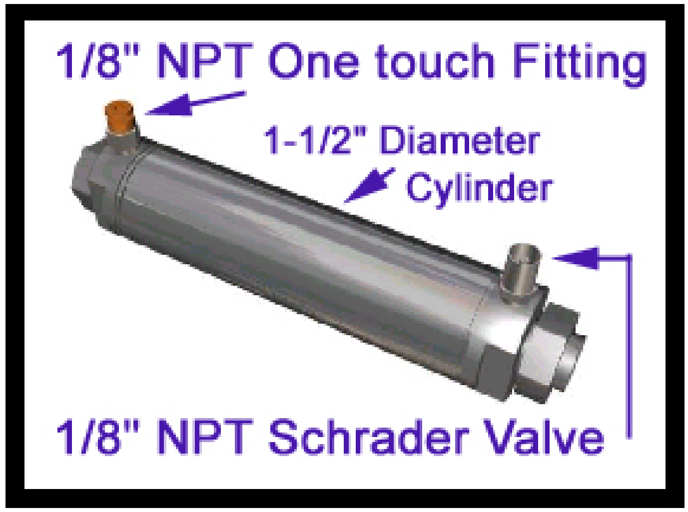


Fig. 1.9

The air reservoir stores the pressurized air used to operate the pneumatic circuit components. The air reservoir acts like a pneumatic battery. Using an understanding of density, and volume, a pair of dial calipers, and some basic CAD skills, it is possible to closely <u>approximate</u> the interior volume of the pneumatic reservoir. Note: the reservoir cannot be dismantled.

National Pipe Thread (NPT) refers to a U.S. standard for tapered (NPT) threads used to join pipes and fittings.

Notebook Exercise: Draw a sketch of the GEARS-IDS™ storage reservoir. Include all the outside dimensions. This data is necessary to determine the interior volume of the reservoir.

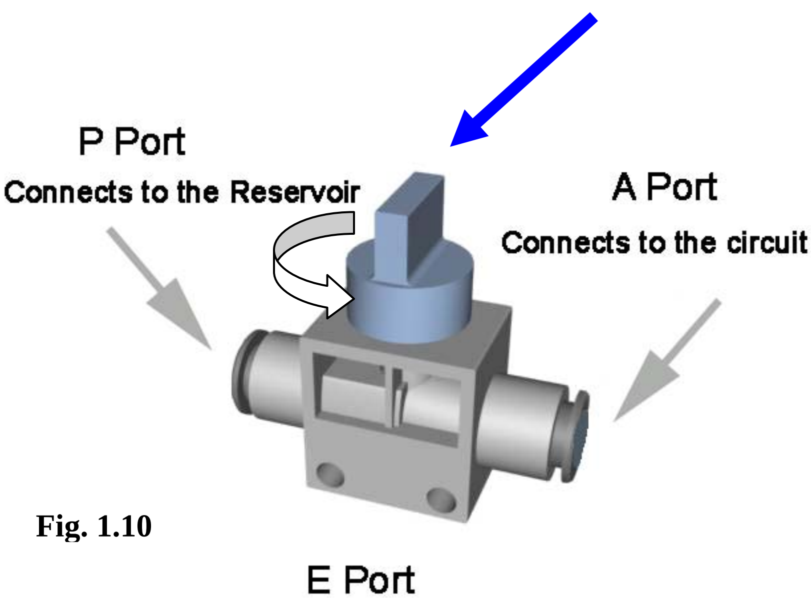


Fig. 1.10

Note: The valve is shown in the off position. The (blue) hand knob is on when it is turned to align with the flow of air.

3-2 Hand Valve

The 3-2 hand valve performs exactly like the 3-2 solenoid. This valve is manually *(hand)* operated and used as an on/off valve for the entire circuit. A 3-2 valve is an essential safety component, because when the valve is closed, the circuit pressure is automatically vented. Automatic venting of the circuit pressure when the air supply is turned off renders the pneumatic circuit safe.

The E port or exhaust port is a small hole in the underside of the valve.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

The GEARS-IDS™ Basic Pneumatic Circuit

Figure 1.11 shows the correct layout and order of components used to make a working pneumatic circuit.

Note: The solenoid valve is actuated through a connection to the PWM switching channel on the GEARS-IDS™ 2 channel speed controller (Not shown).

It is important to observe correct placement of air input and output lines. The 3-2 hand valve, regulator and 3-2 solenoid valve must be plumbed correctly with respect to air input and output lines. Failure to observe the directional arrows or port designations will prevent the circuit from performing correctly.

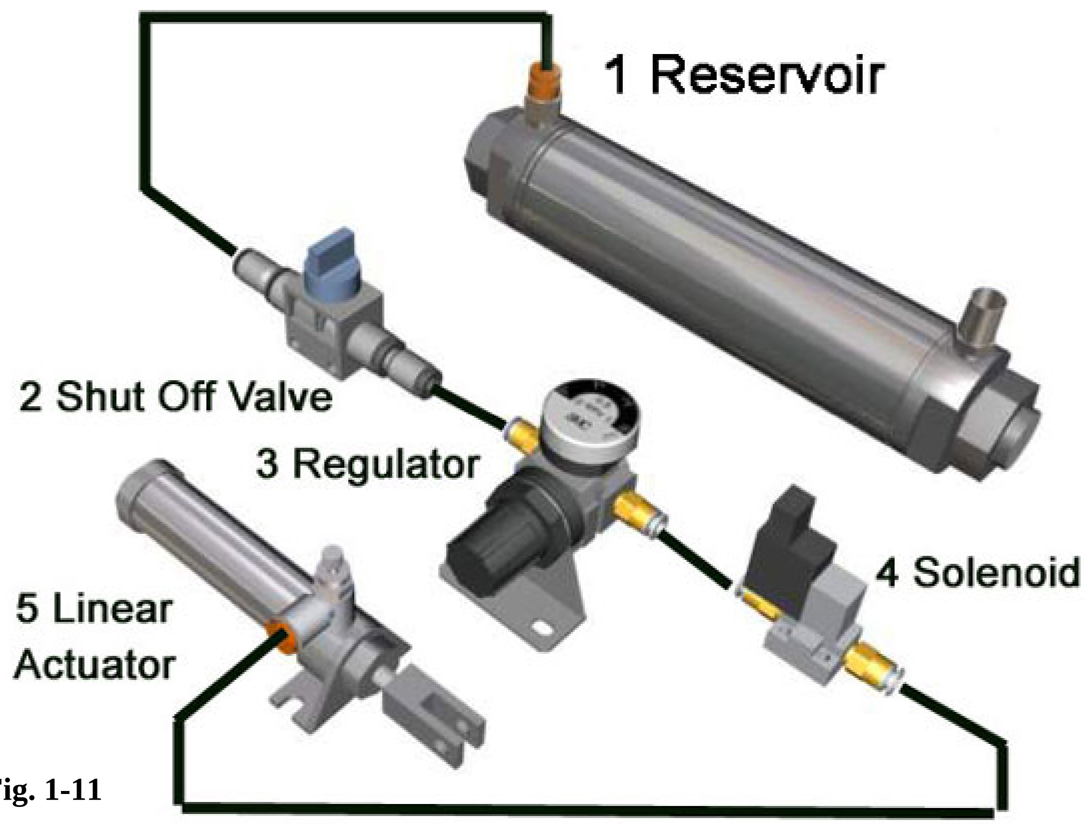


Fig. 1-11

Directions on how to assemble the working pneumatic circuit shown in figure 1-11 are available by clicking on the picture, or by opening the Identify and Assemble Pneumatic Circuits activity sheet.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Pneumatic Schematics

Using Symbols to Design Pneumatic Circuits

Schematics are created using universally accepted diagrams, drawings or symbols to represent elements of a system. Engineers and technical trade's people use symbols to communicate how differing system components can be arranged and integrated.

Symbols and schematics diagrams are a form of "Picture shorthand" used in nearly all technical and engineering fields. Some fields that make use of specialized symbol libraries include electrical, electronic, architectural, pneumatic, piping, and welding to name a few.

Learning to read and use symbols to create schematic representations of pneumatic circuits, provides an opportunity to develop an engineering skill.

The following symbols represent the basic pneumatic circuit components found in the GEARS-IDS™ kit. These symbols plus a few others, can be used to construct schematics of a wide variety of pneumatic circuits. Learn to recognize and use these symbols to evaluate and describe pneumatic circuit designs.

Read the Symbol

The arrow through the spring at the top of the regulator symbol is indicates adjustable regulator control.

The double arrow through the center of the symbol indicates self-relieving capabilities.

Pressure Regulator (symbol)

Adjustable and self-relieving

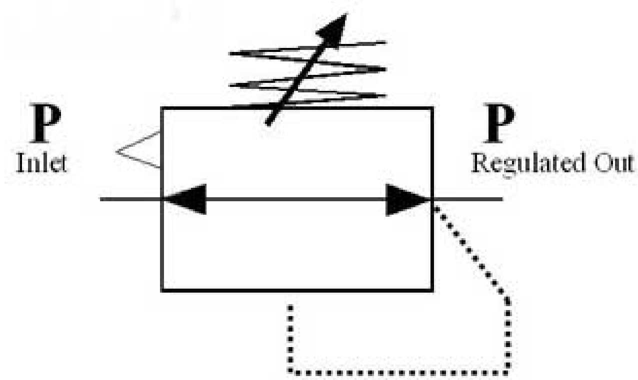


Fig. 1-12

Pressure Regulator

Adjustable and self relieving

Safety and Self-relieving

A self relieving feature provides additional safety because of the added capability of managing under and over pressure conditions downstream of the regulator.

If an overpressure condition is created downstream of the regulator, it will vent the circuit and release the overpressure to the atmosphere.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

3-2 Manual Valve

The small rectangle on the end of the valve symbol indicates manual valve operation.

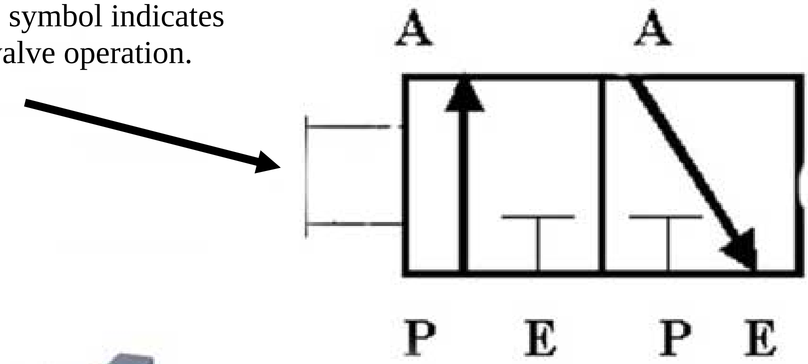
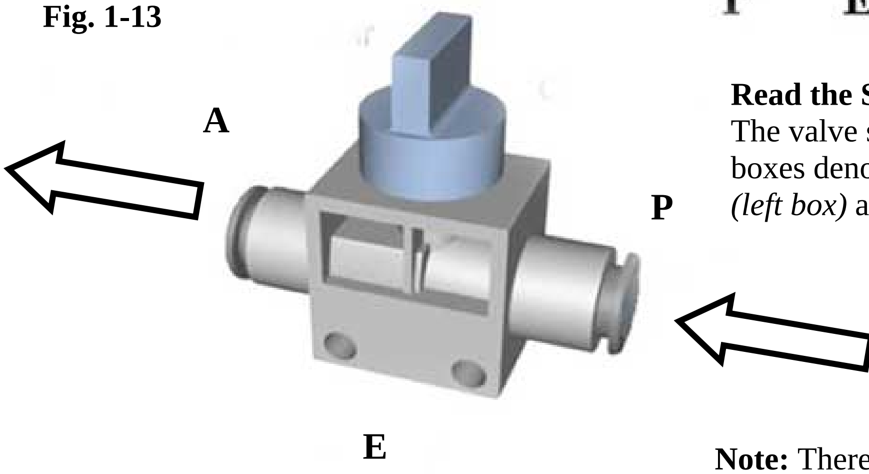


Fig. 1-13



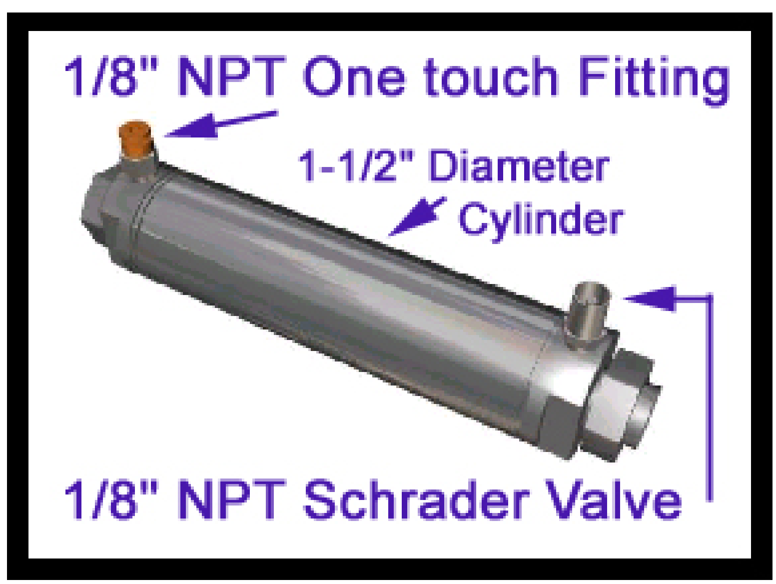
Read the Symbol
The valve symbol is comprised of two boxes denoting two conditions; Passing *(left box)* and not passing *(right box).*

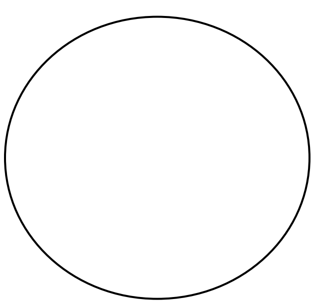
Note: There is always a path for air to flow either to or from the A port. This allows for the A port to be pressurized or exhausted to the atmosphere

The **E** port is a small opening in the underbody of the 3-2 manual valve. This port vents circuit or component air to the atmosphere, allowing the downstream components to be safely depressurized when the circuit is turned off.

Storage Reservoir

The storage reservoir contains a finite volume of pressurized air. The reservoir has two ports; The inlet port is fitted with a one way fill valve called a Schrader valve and the outlet port is fitted with a one touch fitting, used to connect the reservoir to any pneumatic circuit.





Storage Reservoir symbol

Fig. 1-14

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Single Acting Cylinder

Flow valve is not shown in the symbol

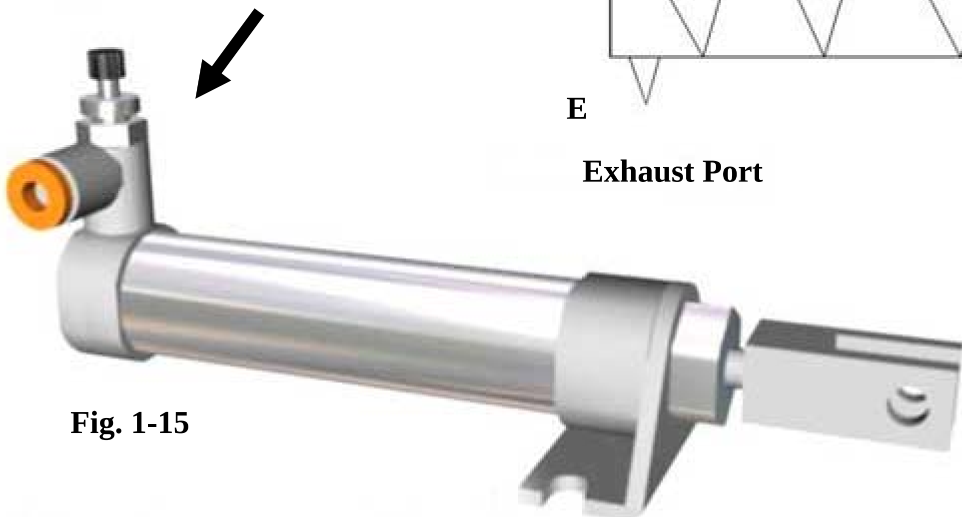
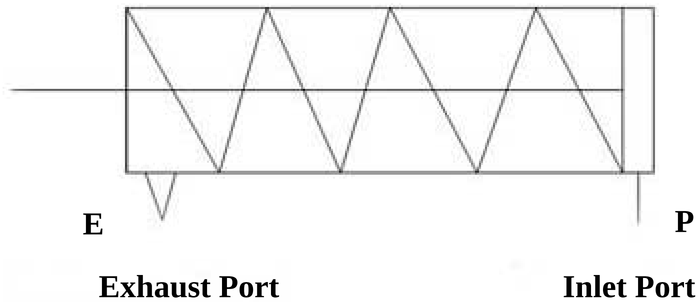
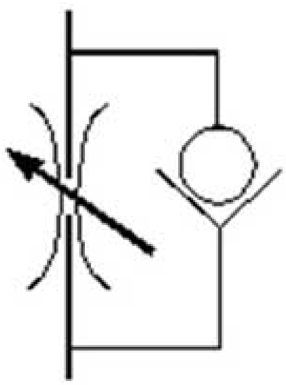


Fig. 1-15

Note the Internal Return Spring





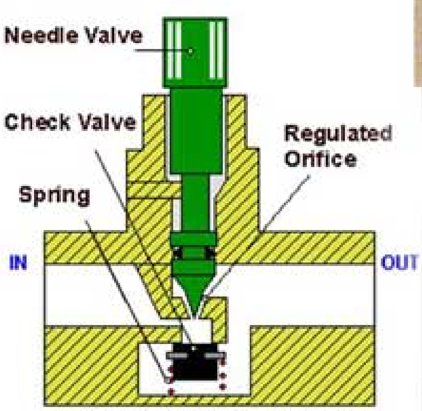


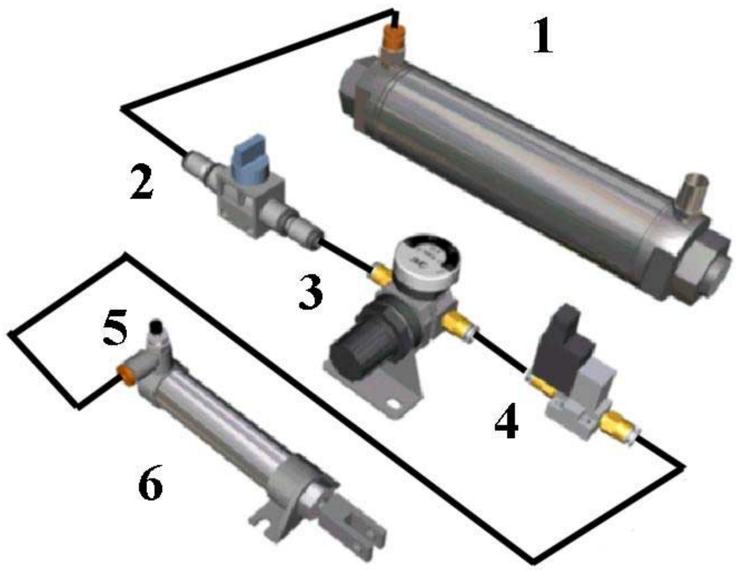
Fig. 1-16

Speed Controller Valves
The speed or flow control valve pictured on the left, regulates the rate at which pressurized air moves from the outlet (out) port to the inlet (in) port. This is accomplished by screwing the needle valve in to slow the flow or out to allow more air to flow past the needle valve. The valve contains a secondary path or circuit that allows air to flow freely and unrestricted from the inlet (in) port to the outlet (out) port

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

GEARS-IDS 6 Pneumatic Circuit Components

Fig. 1-18



1. **Reservoir**
2. **On Off Valve**
3. **Regulator**
4. **Solenoid Valve**
5. **Flow Control Valve**
6. **Pneumatic Cylinder**

Figures 1-18 (above) and 1-19 (on the following page) are similar circuits.

Figure 1-19 is a schematic representation of the pneumatic circuit pictured in figure 1-18.

The schematic illustration on the following page, is constructed using symbols that graphically illustrate the form and function of the pneumatic components.

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

The Symbols Can Be Arranged to Illustrate Complete Circuits

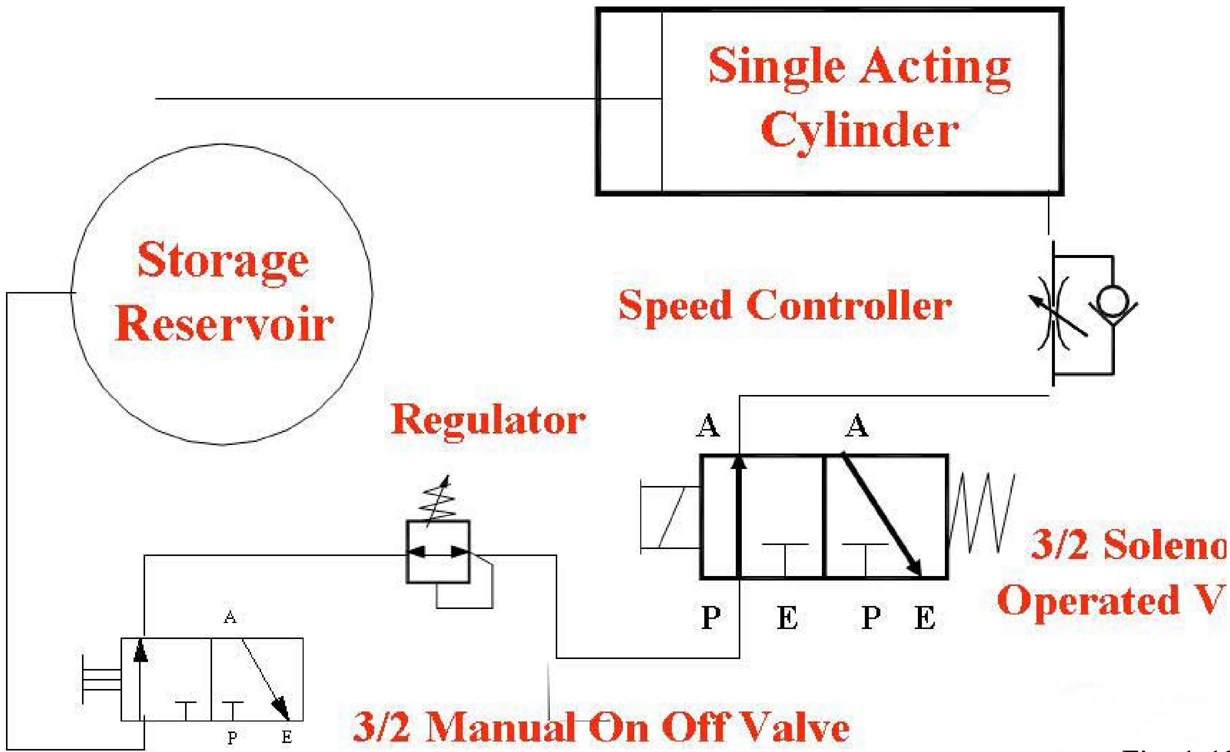


Fig. 1-19

105 Webster St. Hanover Massachusetts 02339 Tel. 781 878 1512 Fax 781 878 6708 www.gearseds.com

Gears Solar Tracker with the BASIC Stamp

Student Guide

DRAFT 031

PARALLAX INC

DISCLAIMER OF LIABILITY

COPYRIGHTS AND TRADEMARKS

TABLE OF CONTENTS

Chapter #1: Limit and Light Sensing Circuits

There are two types of sensing circuits you will need to make the Solar Tracker follow the sun. The first is a simple safety precaution: limit switches to make sure the Solar Tracker doesn't turn too far in either direction. This is important during prototyping and testing because if it does turn too far, it could damage its wiring or pneumatic tubing. The second circuit is the light direction detection circuit. In this chapter you will build and test the circuit and write PBASIC code that gets information from it. In later chapters, you will use the light direction measurements to decide when and how far to adjust the Solar Tracker's position.

Just getting started? New to electronics or programming? Try out the BASIC Stamp Activity Kit from www.parallax.com. With its *What's a Microcontroller* tutorial book, Basic Stamp HomeWork Board and Electronic parts, it's got everything you'll need to get started. The *What's a Microcontroller?* tutorial is a collection of 40 activities. Each activity is just a few pages, and features a lesson in electronics or programming, or sometimes both. Once you've finished *What's a Microcontroller?*, you will have learned the basics and will be able to approach this material with confidence.

ACTIVITY #1: BUILDING AND TESTING PROTOTYPE LIMIT SWITCHES

You will need to mount limit switches that the Solar Tracker platform bumps into before it reaches the limits of its range of motion. This activity uses normally-open pushbuttons to emulate the function of limit switches. They are electrically the same; the main difference is mechanical. The pushbuttons will be sitting on the PCB and they have to be manually pressed when you want to halt rotation, whereas the mechanical limit switches will be mounted so that the Solar Tracker's rotating platform bumps into the switch and presses it before it rotates too far.

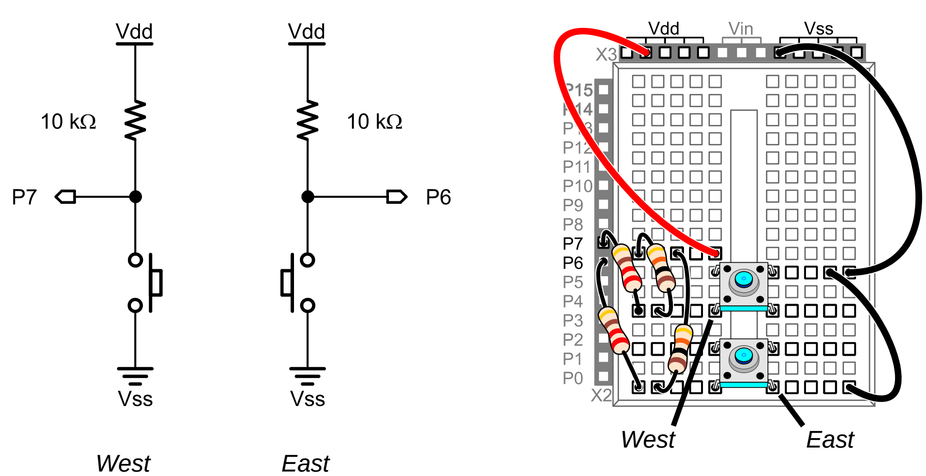
Parts List

A Board of Education with BASIC Stamp 2 connected to your PC running the BASIC Stamp Editor
(2) Pushbuttons – normally open
(2) Resistors – 10 kΩ (brown-black-orange)
(2) Resistors – 220 Ω (red-red-brown)

Schematic

√ Build the circuit shown in Figure 1-1.

Figure 1-1 Circuit for Limit Switch Prototype with Pushbuttons

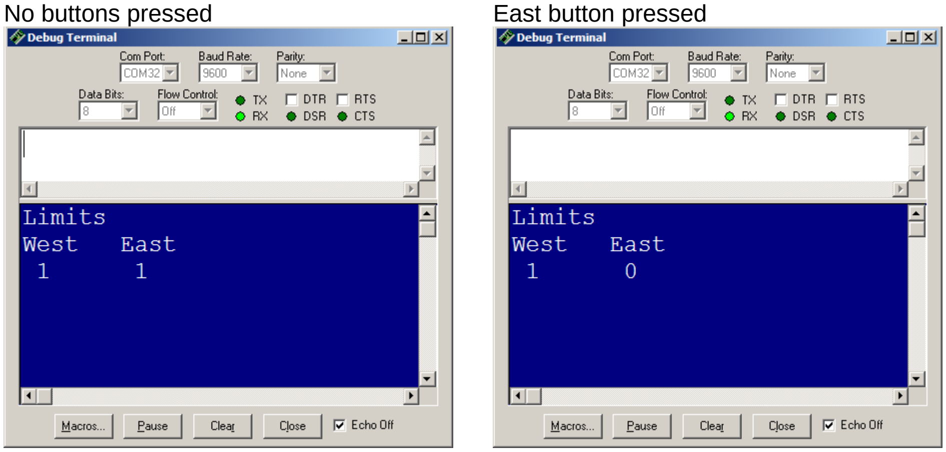


Testing the Pushbuttons

The next example program will display the states of the two pushbuttons. A couple of examples are shown in Figure 1-2. On the left, no buttons are pressed, so both I/O pins receive 5 V, and so their I/O pin aliases (**SWITCH_WEST** and **SWITCH_EAST**) return 1. On the right side of Figure 1-2, the east limit switch was pressed, so the east I/O pin input alias **SWITCH_EAST** changed to zero. Even if you know this code and circuit inside out already, it is still important to test it so that you can make sure your pushbuttons are wired correctly. Otherwise, you run the risk of having the Solar Tracker's platform over- rotate and potentially start yanking on the electrical and pneumatic connections.

Figure 1-2 Limit Switch Display

No buttons pressed East button pressed



Example Program: Test Pushbuttons.bs2

This example program displays the states of the west and east pushbuttons.

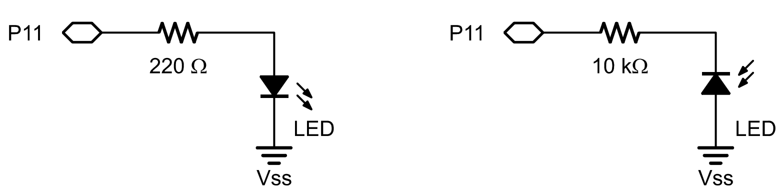
- √ Run the program.
- √ Verify that the west pushbutton circuit (connected to P7) returns a 0 when pressed and a 1 when not pressed.
- √ Repeat for the east pushbutton circuit (connected to P6).

```
' Test Pushbuttons.bs2
' Test prototypes for east and west limit switches.

' {$STAMP BS2}
' {$PBASIC 2.5}

SWITCH_WEST    PIN     7
SWITCH_EAST    PIN     6

PAUSE 1000

DO

  DEBUG HOME, "Limits", CR, "West   East", CR
  DEBUG CRSRX, 1, BIN1 SWITCH_WEST, CRSRX, 8, BIN1 SWITCH_EAST
  PAUSE 50

LOOP
```

ACTIVITY #2: DETECTING LIGHT LEVEL WITH AN LED

The LED circuit on the left of Figure 1-3 can be used to emit light. When current passes through the circuit, the LED emits light. The circuit on the right can be used to measure light. When light strikes the silicon inside the LED, it conducts current. Unlike LEDs, photodiodes are devices specifically designed to collect light (instead of emit it). Although LED's are not specifically designed for this type of application, you will soon see that the BASIC Stamp can be used to make them very sensitive light detectors.

Figure 1-3 Light Emitter and Detector Schematics

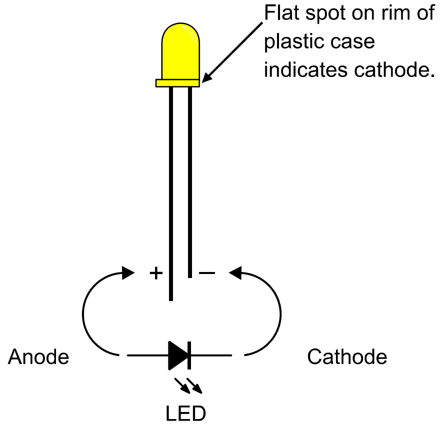


> **For more information on light detection with a photodiode,** download Applied Sensors from www.parallax.com, and consult Chapter 4. Unlike the detection circuit in Applied Sensors, the detection circuit here does not have a capacitor. It instead relies on a capacitance inherent to all diodes, called junction capacitance, to store charge. This is the capacitance of the junction between silicon with two different impurities inside the LED. As current crosses this junction in an LED, the electrons emit light. If light strikes the silicon, it forces electrons across the junction, creating current.

Emitting Light vs. Measuring Light

Take a close look at Figure 1-3. What's the difference between the LED in the two circuits? Notice that its polarity has been reversed. Use Figure 1-4 as a reference when building LED circuit's from schematic.

Figure 1-4 LED Cathode and Anode



Light Emitter Test

This quick tests verifies that your LED is good, and it also demonstrates the LED as a "light emitting diode".

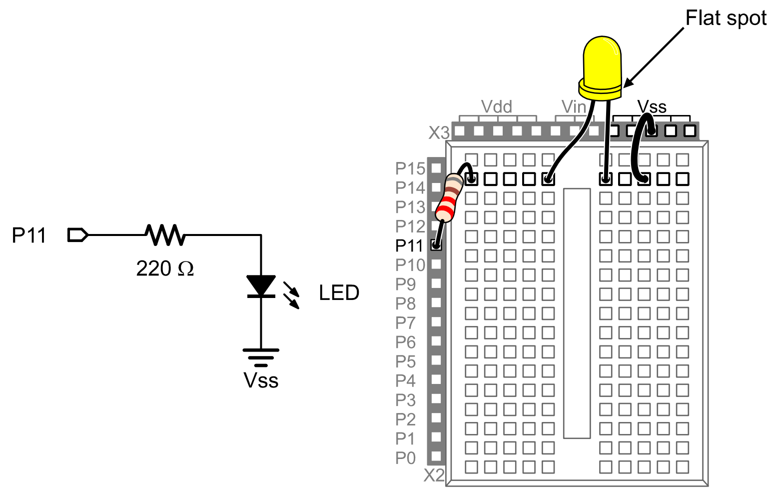
Parts List

(1) LED – yellow
(1) Resistor – 220 Ω (red-red-brown)
(1) Jumper wire - black

LED Light Emitter Circuit

Figure 1-5 shows an LED emitter test circuit. Even if you are already familiar with this circuit, it's still a good idea to build and test it just to make sure your LED works.

Figure 1-5 Light Emitter Schematic and Wiring Diagram



Example Program: Test LED Light Emitter.bs2

This simple example program just turns the LED light on/off about 5 times per second.

√ Load the program and verify that the LED emits light and then turns off at about 5 times per second.

```
' Test LED Light Emitter.bs2

' {$STAMP BS2}
' {$PBASIC 2.5}

DO

  HIGH 11
  PAUSE 100
  LOW 11
  PAUSE 100

LOOP
```

Light Measurements Test

Next, let's rewire the LED circuit for light detection.

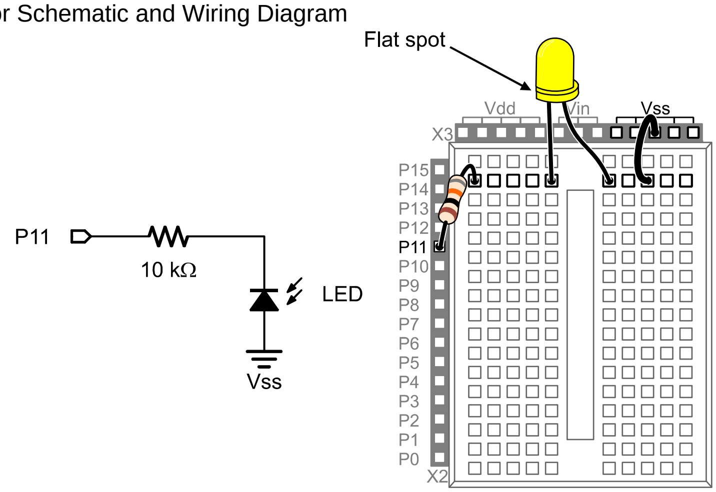
Parts List

(1) LED – yellow
(1) Resistor – 10 kΩ (brown-black-orange)
(1) Jumper wire - black

LED Light Detector Circuit

Figure 1-6 shows an LED light detection test circuit. Notice that the anode and cathode terminals have been swapped, so you'll need to unplug your LED, give it a half turn, and plug it back in.

Figure 1-6 Light Detector Schematic and Wiring Diagram



Example Program: Test LED Light Detector.bs2

This example program applies a **HIGH** voltage (5 V) to the LED and charges up the capacitor that exists at the junction between the two types of silicon sandwiched together in the LED. Then, the **RCTIME** command changes the I/O pin to input (doesn't send a high or low signal, just listens), and measures the time it takes the junction charge to decay below 1.4 V. This time is controlled by how much light shines on the LED. The program measures the number of 2 µs time increments it took for the LED voltage to reach 1.4 V and stores it in a variable named **time**.

- √ Run this program in a fairly well-lit room.
- √ Try casting various levels of shade on the LED and observe how the measurement increases, up to a point.

The largest number of 2 µs time increments the **RCTIME** command can count is 65535, which is 131.07 ms. If it takes longer than that for the diode junction voltage to decay, the **RCTIME** command stores 0 in the time variable.

- √ Increase the amount of shade you cast on the LED and see how close you can get your light measurement to 65535.
- √ Cup your hand over the LED. Can you get it to display 0 when it's too dark?

```
' Test LED Light Detector.bs2

' {$STAMP BS2}
' {$PBASIC 2.5}

time VAR Word

DO

  HIGH 11
  PAUSE 1
  RCTIME 11, 1, time
  DEBUG HOME, DEC5 ?  time
  PAUSE 200

LOOP
```

Your Turn

You can display the number of milliseconds (ms) the **RCTIME** command takes by dividing time by 1000 with the / operator.

√ Modify the program so that it displays the decay time in ms.

You can make you're an **RCTIME** command for slower decay measurements that take too long for **RCTIME**. This involves replacing the **RCTIME** command with an **INPUT** command that changes the I/O pin to an input. This is followed by a loop that counts the number of repetitions until the voltage at the I/O pin decays to below 1.4 V (the I/O pin's threshold between 1 and 0).

√ Comment the **RCTIME** command by placing an apostrophe to the left of it and then add this code just below the commented **RCTIME** command.

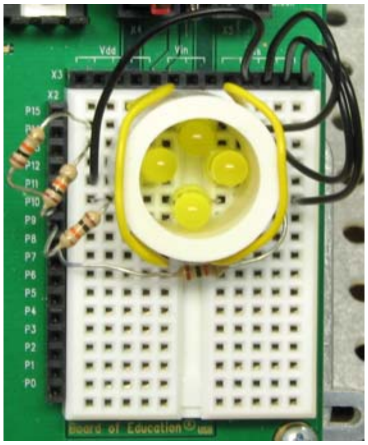
```
time = 0
INPUT 11
DO UNTIL IN11 = 0
  PAUSE 0
  time = time + 1
LOOP
```

√ Try casting the same level of shade that you used to cause the **RCTIME** command to time out and return 0. Now what measurement do you get?

ACTIVITY #3: BUILDING AND TESTING LIGHT DIRECTION SENSOR

Your light direction sensor will be four LEDs inside a section of PVC sprinkler pipe. The tops of the LEDs are below the top of the sprinkler pipe. Depending on where the light source is, the sprinkler pipe casts a shadow on some LEDs while others get full light. The BASIC Stamp will be programmed to get light measurements from each LED with the **RCTIME** command and compare them to figure out the direction of the light source.

Figure 1-7: Light Direction Sensor



Parts List

(4) Resistors – 10 kΩ
(4) LEDs – Yellow
(4) Jumper wires – Black
(2) Additional jumper wires
(1) ¾" Length of ½" diameter white PVC pipe
60 W desk lamp

Schematic and Wiring

Figure 2-2 shows a schematic of the light detector array, and Figure 2-3 shows the recommended arrangement and wiring. Remember as you build this circuit that each resistor should connect to the LED's cathode, which is the terminal by the flat spot on the LED's plastic case. Each LED's anode gets connected to ground (Vss).

Figure 1-8 Light Detector Schematic

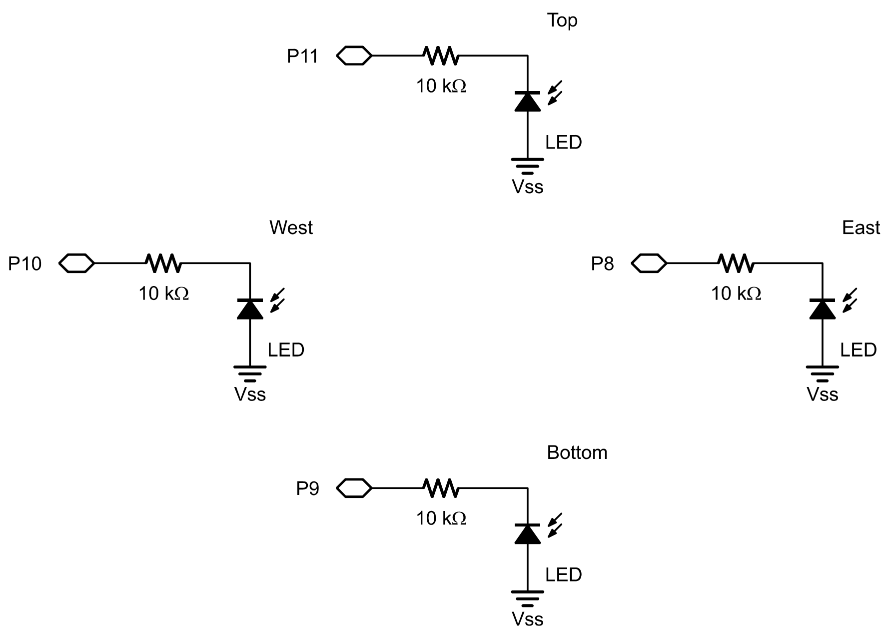
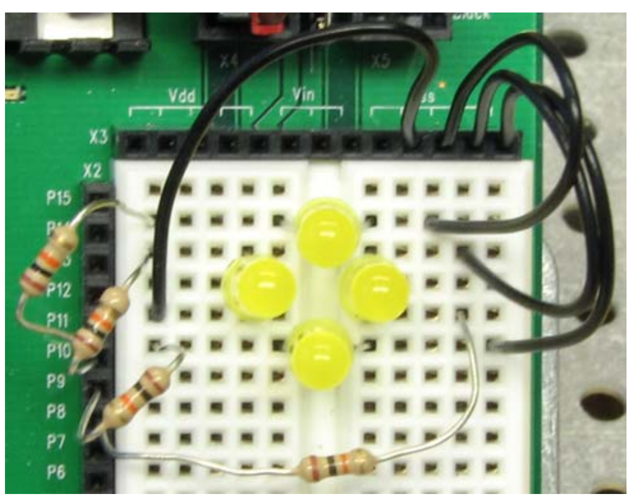


Figure 1-9 Light Detector Recommended Circuit Placement



Differences in Light Level Measurements Indicate Direction

If a light source is directly above the section of PVC pipe in Figure 2-1, it won't cast any shadows, and all the light sensors should return roughly the same measurements. As soon as the light source moves off-center, the PVC cylinder will cast shadows on one or more LEDs.

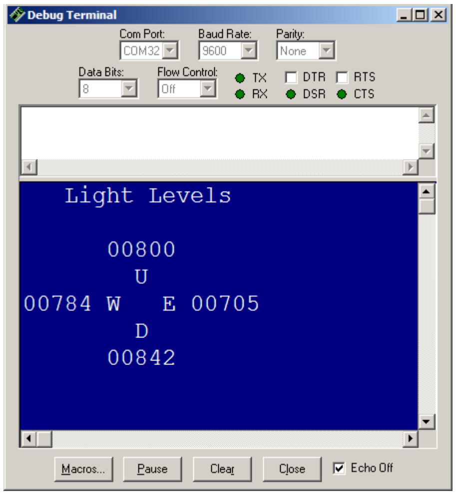
For example, if the light source moves west, but doesn't change its vertical position, a shadow will start to block direct light from shining on the west LED, but it will still get to the rest. The west light detector will start to return a higher number (less light), while the east one will stay about the same as it was. If the source of light instead travels west and downward LEDs, the PVC cylinder will cast shadows on the west and bottom LEDs. Their measurements will go up because they are getting less light, while the east and top LED measurements will stay about the same as they were.

Figure 1-10 shows the Debug Terminal display from the next example program with a desk lamp shining almost directly into the PVC cylinder from about two feet above. The LEDs indicate that it might be slightly down and to the west, but only slightly. If the bulb is dimmer or was held further away, all the values would be larger. If it is instead brighter or closer, all the values would be smaller. When you move the lamp up, or down or sideways, or some combination of right/left and up/down, you'll see differences in the E/W and U/D values that your programs can use to figure out the direction.

You can also take it outside and try it with the sun on a clear day. If you orient the Solar Tracker so that sun is shining straight into the tube, all the measurements will be fairly similar. If you orient it so that the sun is not shining directly into the tube, you will see differences between the E/W and U/D light sensors that indicate which direction the sun is in relation to the light direction sensor.

Figure 1-10: Debug Terminal Light Measurements

Smaller values indicate more light; larger values indicate less light.



Example Program: Test Light Direction Sensor.bs2

This program will allow you to measure the response of the light direction sensor to the position of a point light source such as a desk lamp in an otherwise low-light room, or the sun.

- √ Load and run the program.
- √ Start with a 60 W desk lamp. Point the bulb directly at the light sensor from 2 to 4 feet directly above the sensor, and record the measurements.

√ Next, try pointing the desk lamp bulb at the Solar Tracker from various points east, west, above and below the direction sensor. (Assume the platform is facing south and you are facing north looking at it. East will be to your right, and west to your left.)
√ Record the measurements at each point and explain what happens.
√ Try the same distances and angles with a 40 W bulb. What's the main difference?

```
' Test Light Direction Sensor.bs2
' Displays measurements of LED light detector array.
' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ Declarations ]-----------------------------------------------------

LED_TOP           CON     11
LED_WEST          CON     10
LED_BOTTOM        CON     9
LED_EAST          CON     8

LIGHT_TOP         CON     3
LIGHT_WEST        CON     2
LIGHT_BOTTOM      CON     1
LIGHT_EAST        CON     0

time              VAR     Word(4)
index             VAR     Nib
pindex            VAR     index

x                 VAR     Nib
y                 VAR     Nib

'-----[ Initializataion ]--------------------------------------------------

PAUSE 1000

DEBUG CLS, "   Light Levels",
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"

'-----[ Main ]-------------------------------------------------------------

DO

  GOSUB Get_Decay_Times
  GOSUB Display_Decay_Times
  PAUSE 500

LOOP

'-----[ Subroutine Get_Decay_Times ]------------------------------------------

Get_Decay_Times:
  FOR pindex = LED_EAST TO LED_TOP
    HIGH pindex
    PAUSE 1
    RCTIME pindex, 1, time(pindex - LED_EAST)
  NEXT
  RETURN

'-----[ Subroutine Display_Decay_Times ]--------------------------------------

Display_Decay_Times:
  FOR index = LIGHT_EAST TO LIGHT_TOP
    LOOKUP index, [12, 6, 0, 6], x
    LOOKUP index, [4, 6, 4, 2], y
    DEBUG CRSRXY, x, y, DEC5 time(index), CR
  NEXT
  DEBUG CR, CR
  RETURN
```

How Test Light Direction Sensor.bs2 Works

The Declarations section has four constant declarations for LED I/O pins: **LED_TOP**, **LED_WEST**, etc. These constants were used instead of pin directives because they will get used in a **FOR…NEXT** loop, and **PIN** directives will return the measured value at the I/O pin in that circumstance. The next four declarations starting with **LIGHT_TOP** are index values for entries in a variable array that stores light measurements. That array is declared with **time VAR Word(4)**. A nibble size **index** variable is also declared, followed by **pindex**, which is an alias of **index**. In other words, **index** and **pindex** are two different names for the same nibble in the BASIC Stamp's memory. Nibbles named **x** and **y** are also declared for positioning the cursor on the Debug Terminal.

```
'-----[ Declarations ]-------------------------------------------------

LED_TOP          CON     11
LED_WEST         CON     10
LED_BOTTOM       CON     9
LED_EAST         CON     8

LIGHT_TOP        CON     3
LIGHT_WEST       CON     2
LIGHT_BOTTOM     CON     1
LIGHT_EAST       CON     0

time             VAR     Word(4)
index            VAR     Nib
pindex           VAR     index

x                VAR     Nib
y                VAR     Nib
```

After a 1 second delay, the **DEBUG** command displays **"   Light Levels"**, followed by cursor placements with the **CRSRXY** formatter to position the U, D, W, and E characters. **CRSRXY** has to be followed by **x**, the number of spaces over, and **y**, the number of carriage returns down.

```
'-----[ Initializataion ]----------------------------------------------

PAUSE 1000

DEBUG CLS, "   Light Levels",
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"
```

The Main routine calls the **Get_Decay_Times** subroutine, then the **Display_Decay_Times** subroutine. Then, it delays for ½ s before repeating in a **DO…LOOP**.

```
'-----[ Main ]---------------------------------------------------------

DO

  GOSUB Get_Decay_Times
  GOSUB Display_Decay_Times
  PAUSE 500

LOOP
```

The **Get_Decay_Times** subroutine uses a **FOR…NEXT** loop to measure the RC decay time of each LED circuit. Since **LED_EAST** is 8, and **LED_TOP** is 11, the **FOR…NEXT** loop starts by setting **pindex** to 8, and it repeats through **pindex = 11**. The first time through, it sets P8 high, waits 1 ms, and does an **RCTIME** measurement on P8. Since **pindex** stores 8, and **LED_EAST** is 8, the result of **pindex – LED_EAST** is 0. So the result of the **RCTIME pindex, 1, time(pindex – LED_EAST)** measurement gets stored in **time(0)**. The second time through the loop, **pindex** is 9, so the **RCTIME** measurement gets performed on the LED circuit connected to

P9. The **RCTIME** result gets stored in **time(1)** because **pindex** is now 9, but **LED _EAST** is still 8. Two more times through the loop take **RCTIME** measurements on P10 and P11 and store the results in **time(2)** and **time(3)** respectively.

```
'-----[ Subroutine Get_Decay_Times ]-----------------------------------

Get_Decay_Times:

  FOR pindex = LED_EAST TO LED_TOP
    HIGH pindex
    PAUSE 1
    RCTIME pindex, 1, time(pindex - LED_EAST)
  NEXT
  RETURN
```

The **Display_Decay_Times** subroutine assumes that four elements in the **time** array have been loaded with RC decay measurements. Its **FOR…NEXT** loop indexes from **LIGHT_EAS**T, which is 0, to **LIGHT_TOP**, which is 3. Two **LOOKUP** commands use the **index** variable to select values from their lookup tables and store them in **x** and **y** variables. For example, if **index** is 0, the first **LOOKUP** command stores 12 in **x**, if **index** is 1, it stores 6 in **x**, and so on. After the **LOOKUP** commands have stored cursor placement values in the **x** and **y** variables, **DEBUG x, y, DEC5 time(index)** places the cursor at a location in the Debug Terminal, and then displays the 5 digit decimal value stored in **time(index)**.

```
'-----[ Subroutine Display_Decay_Times ]-------------------------------

Display_Decay_Times:

  FOR index = LIGHT_EAST TO LIGHT_TOP
    LOOKUP index, [12, 6, 0, 6], x
    LOOKUP index, [4, 6, 4, 2], y
    DEBUG CRSRXY, x, y, DEC5 time(index), CR
  NEXT
  DEBUG CR, CR
  RETURN
```

The first time through the **Display_Decay_Times** subroutine's **FOR…NEXT** loop, **index** stores 0, so the first **LOOKUP** command stores 12 in **x** and the second **LOOKUP** command stores 4 in **y**. Next, **DEBUG CRSRXY, x, y, DEC5 time(index)** places the cursor 12 spaces over, and 4 carriage returns down from the top-left corner of the terminal. Then, it displays the 5 digit decimal contents of **time(0)**. The second time through the loop, **index** stores 1. So the **LOOKUP** commands store 6 in both the **x** and **y** variables. The **DEBUG** command then displays the contents of **time(1)** at 6 spaces over and 6 spaces down in the Debug Terminal.

Chapter #2: Motor and Piston Control

BEFORE YOU START

Before starting here, complete the activities in these two PDF documents available from www.gearseds.com

- solar_tracker_const_guide_rev3.pdf
- GEARS II Speed Controllersrev8.pdf

ACTIVITY #1: TRACKER MOTOR CONTROL

The motor makes the Solar Tracker platform rotate from east to west so that the panel follows the sun as the day goes by. In this activity, you will experiment with some code that makes the Solar Tracker platform rotate east to west, and then back again.

Figure 2-1 shows the Solar Tracker with Board of Education mounted on the Solar Tracker's platform.

Figure 2-1: Board of Education Mounted on Solar Tracker Platform

Draft note: Revise based on final supply configuration.

Parts Required

- Board of Education and BASIC Stamp 2 microcontroller
- GEARS Solar Tracker Mechanical Hardware
- GEARS IDS II Speed Controller with Integrated Valve Control

Connections to the Board of Education (Review)

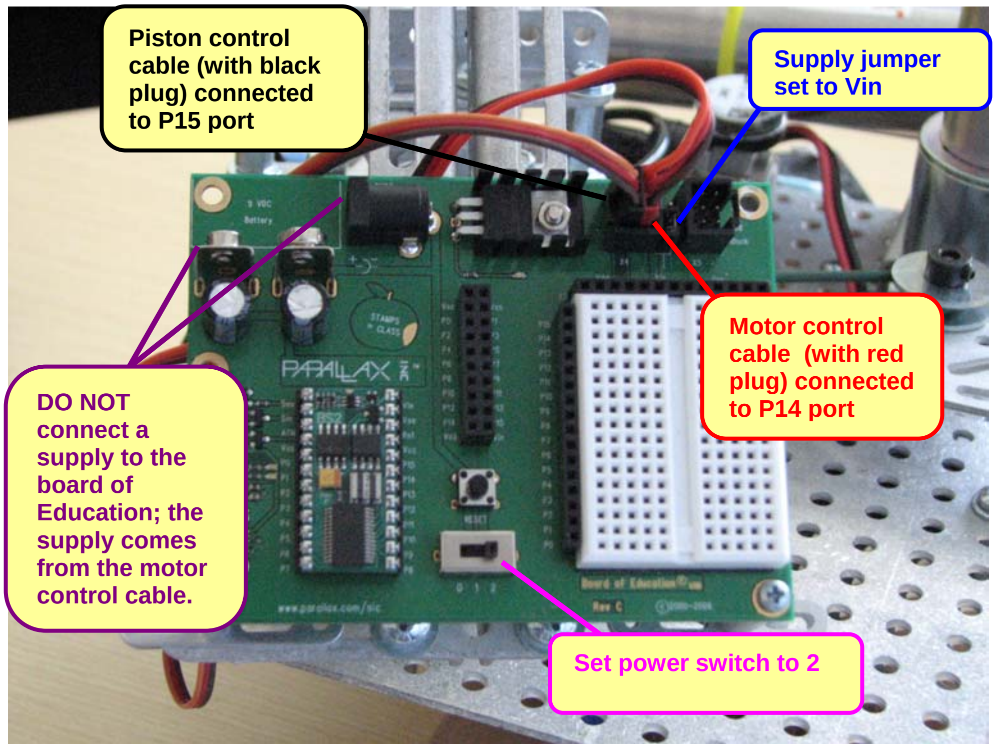
Figure 2-2 reviews the connections you made between the Board of Education and the GEARS IDS II Speed Controller with Integrated Valve Control.

√ The IDS II controller's piston control signal cable is connected to the Board of Education's P15 servo port. This cable typically has a black plug.

√ The IDS II controller's motor control cable is connected to the Board of Education's P14. This cable typically has a red plug.

√ DO NOT CONNECT POWER TO THE BOARD OF EDUCATION'S POWER INPUTS. Draft note: Pending final power supply design.
√ The power jumper between the X4 and X5 servo headers is set to Vin.
√ When the Board of Education's power switch gets set to 2, it will get its power supply from IDS II controller via the motor and piston control cables.

Figure 2-2: Board of Education and BASIC Stamp Mounted on Solar Tracker Hardware



Motor Control Test Program

The example programs in this chapter are designed so that Solar Tracker motor and piston control and other functions such as sensor monitoring and display are all handled in subroutines. Most of the coding you will be doing is in the Main routine. In the case of motor and piston control and sensor display, your Main routine will set variable values and then call subroutines. The subroutines will do a job based on the values of the variables you have set. In later activities, your Main routine will also call subroutines that read sensors and store the measurements in a series of variables (the time array) for your code in the Main routine to use.

Review of Motor Control Signals for the IDS II:

- Control pulses have to be sent to the IDS II at least once every 500 ms. If the time between pulses is longer than that, you'll have to reset power to the IDS II controller to restart it.
- Full speed clockwise motor signal - a pulse that lasts 1 ms (**`PULSOUT 14, 500`**)
- Full speed counterclockwise signal - a pulse that lasts 2 ms (**`PULSOUT 14, 1000`**)
- Stop signal – a pulse that lasts 1.520 ms (**`PULSOUT 14, 760`**)
- For speed control, **`PULSOUT`** values closer to 760 make the motor turn slower, and values further from 760 (but not to exceed 500 or 1000) will make the motor turn faster.

The next example program has a subroutine named **Motor_Piston_Control** that uses a **PULSOUT** command to deliver a pulse to the motor control line, which is connected to BASIC Stamp I/O pin P14 (via servo port 14 on the Board of Education). It uses a variable named **motorPulse** to determine the control pulse it sends to the IDS II for motor control. This variable can be set to 500 for full speed clockwise, 1000 for full speed counterclockwise, and 760 for making it stop. Values closer to 760 will slow the motor down, and values closer to 500 or 1000 will make the motor turn faster.

After you set the **motorPulse** variable, all you have to do is call the **Motor_Piston_Control** subroutine at least once every 500 ms. Since the code takes some time to run, and the subroutine has a 20 ms pause built-in, you might want to make that 470 ms or less to be on the safe side.

So, to control motor speed, just set the **motorPulse** variable and then call the **Motor_Piston_Control subroutine**. How about controlling how long it makes the motor turn? The way the next example program does it is to assume that each subroutine call takes about 1/40 of a second. So, if you want to send the IDS II a signal that instructs it to make the motor turn for 5 seconds, call the **Motor_Piston_Control** subroutine 200 times.

$$5 \text{ seconds} \times \frac{40 \text{ pulses}}{\text{second}} = 200 \text{ pulses}$$

Here's an excerpt from the next example program's Main routine that instructs the Solar Tracker to turn east (assuming it's facing south). This code sets the **motorPulse** variable to 900 (about 60 % of full speed counterclockwise). Then, it calls the **Motor_Piston_Control** subroutine 150 times. You can assume that each time through the loop takes about 1/40 of a second, so the IDS II speed controller gets just under 4 seconds of signal to make the motor turn counterclockwise at 60% of full speed.

```
motorPulse = 900                  ' Rotate East

FOR counter = 1 TO 150            ' for about 3 seconds.
  GOSUB Motor_Piston_Control
NEXT
```

Here's another excerpt from the next example program. It also makes the Solar Tracker's motor turn at about 60 % of full speed, but this time, the motor will turn counterclockwise instead of clockwise, and the Solar Tracker's platform will rotate toward the west (again, assuming it's facing south).

```
motorPulse = 600                  ' Rotate West

FOR counter = 1 TO 150            ' for about 3 seconds
  GOSUB Motor_Piston_Control
NEXT
```

Here is one last excerpt from the next example program. It's at the end of the Main routine, and it sets the motor to stop/neutral and then calls the **Motor_Piston_Control** subroutine indefinitely. Why not just stop calling the **Motor_Piston_Control** subroutine? As mentioned earlier, if you do that, you'll have to power to the IDS II controller by shutting off the power and then turning it back on.

```
motorPulse = 760                  ' Set motor to neutral

DO                                ' Piston & motor neutral indefinitely
  GOSUB Motor_Piston_Control
LOOP
```

Example Program: Basic Motor Control.bs2

This next example sends a turn counterclockwise at 60 % of full speed signal to the IDS II controller for about 3.75 seconds, followed by a clockwise at 60 % of full speed signal for 3.75 seconds, followed by a stop signal that lasts indefinitely. Assuming the platform is in the center of its range of motion, and facing south, the program will make its platform rotate east briefly, then west briefly, and then stop.

In addition to the code examples just discussed, the Main routine has **DEBUG** commands that display the value of **motorPulse** each time it gets changed.

- √ Make sure that the platform is near the center of its range of motion before running the program.
- √ Open Basic Motor Control.bs2 into the BASIC Stamp Editor
- √ Turn on Solar Tracker.
- √ Load the program into the BASIC Stamp. (Click the BASIC Stamp Editor's Run button or select Run from the Run menu.)
- √ Verify that the Solar Tracker rotates slowly east, then slowly west, then stays still.
- √ Press and release the Board of Education's Reset button to restart the program and repeat the action.

```
' Basic Motor Control.bs2
' Test Solar Tracker motor control.  This program sends a rotate
' counter clockwise at 60 % of full speed for 3.75 seconds, followed by
' a corresponding clockwise signal, followed by an indefinite full-stop.


' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ Declarations ]-----------------------------------------------------

MOTOR           PIN     14                      ' Red header
PISTON          PIN     15                      ' Black header

PULSE_DELAY     CON     20                      ' 20 ms between pulses

motorPulse      VAR     Word                    ' Motor control pulse duration
pistonPulse     VAR     Word                    ' Pistion control pulse duration
counter         VAR     Byte                    ' Counting variable

'-----[ Initializataion ]--------------------------------------------------

GOSUB ST_Controller_Init                        ' Required, takes ~ 4 s.

'-----[ Main ]-------------------------------------------------------------

motorPulse = 900                                ' Rotate East
DEBUG ? motorPulse

FOR counter = 1 TO 150                          ' for about 3.75 seconds.
  GOSUB Motor_Piston_Control
NEXT

motorPulse = 600                                ' Rotate West
DEBUG ? motorPulse

FOR counter = 1 TO 150                          ' for about 3.75 seconds
  GOSUB Motor_Piston_Control
NEXT

motorPulse = 760                                ' Set motor to neutral
DEBUG ? motorPulse

DO                                              ' Piston & motor neutral indefinately
  GOSUB Motor_Piston_Control
LOOP

'-----[ Subroutine Motor_Piston_Control ]------------------------------------

' Must call at least every 500 ms.

Motor_Piston_Control:

  PULSOUT MOTOR, motorPulse                     ' Motor control pulse
```

```
  PULSOUT PISTON, pistonPulse                ' Pistion control pulse
  PAUSE 20

  RETURN

'-----[ Subroutine ST_Controller_Init ]--------------------------------------

' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on channels,
' pause antoher 1/10 s, then neutral pulses for ~ 2 s so that motor/valve
' controller gives BS2 control.

ST_Controller_Init:

  motorPulse = 760                             ' Neutral pulse durations.
  pistonPulse = 760

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  Return
```

Your Turn – Controlling Speed and Rotation Angle

In the program's Main routine, the value of **motorPulse** controls speed, and the number of times the **Motor_Piston_Control** subroutine gets called controls the amount of time the signal gets sent, which in turn controls the run time.

> **DO NOT USE LARGE VALUES THAT MAKE THE TRACKER PLATFORM ROTATE TOO FAR!** If the tracker platform rotates too far, it will pull apart the pneumatic lines, and then you'll have a lot of repair work to do. The limit switches will not be incorporated into motion control until Chapter 3.

- √ Experiment with different values of **motorPulse = _____** and **FOR counter = 1 to _____**.
- √ Modify the program so that it turns to face east, and then turns about 1/12th of the way back toward west each 10 seconds for 120 seconds. This will emulate an hourly position update during a 12 hour day. The Solar Tracker should then reset back to pointing east, and wait for the next day to start (120 seconds later).

ACTIVITY #2: TRACKER PISTON CONTROL

The Solar Tracker's piston controls whether the platform is facing low or high in the sky. In this activity, you will test piston control with a modified Main routine. The rest of the example program (variables, subroutines, etc) will be the same.

Controlling the Piston

The **Motor_Piston_Control** subroutine also sends piston control signals to the IDS II controller. All you have to do is set a variable named **pistonPulse** before calling **Motor_Piston_Control**. You can set **pistonPulse** to 960 to let pressurized air into the cylinder and extend the piston rod, or 760 to release air from the cylinder and allow the built-in spring to retract the rod. You can also control the amount of time the piston is in a given position the same way you just controlled motor run time, by calling the **Motor_Piston_Control** subroutine in a loop. (Again, assume 40 pulses per second.)

Here's a Main routine excerpt from the next example program (below). It sets the **pistonPulse** variable to 960 (cylinder pressurized, piston rod extended, then waits for about 3.75 seconds with a **FOR…NEXT** loop that calls the **Motor_Piston_Control** subroutine 150 times. Then, it sets the **pistonPulse** variable to 760 (release air, piston rod retracts), and the **DO…LOOP** that follows calls the **Motor_Piston_Control** subroutine indefinitely.

```
'-----[ Main ]--------------------------------------------------------

pistonPulse = 960               ' Set piston to push plate forward

FOR counter = 1 TO 150          ' for about 3.75 seconds
  GOSUB Motor_Piston_Control
NEXT

pistonPulse = 760               ' Set piston to let go

DO                              ' Piston & motor neutral indefinitely
  GOSUB Motor_Piston_Control
LOOP
```

Example Program: Basic Piston Control.bs2

This program tilts the Solar Tracker's platform forward briefly so that it faces lower in the sky. Then, it releases the platform, so that it faces higher in the sky again.

- √ Pump up the air tank.
- √ Open the manual valve.
- √ Turn on power to the Solar Tracker.
- √ Load Basic Piston Control.bs2 into the BASIC Stamp.

```
' Basic Piston Control.bs2

' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ Declarations ]----------------------------------------------------

MOTOR          PIN     14                     ' Red header
PISTON         PIN     15                     ' Black header

PULSE_DELAY    CON     20                     ' 20 ms between pulses

motorPulse     VAR     Word                   ' Motor control pulse duration
pistonPulse    VAR     Word                   ' Pistion control pulse duration
counter        VAR     Byte                   ' Counting variable

'-----[ Initializataion ]-------------------------------------------------

GOSUB ST_Controller_Init                      ' Required, takes ~ 4 s.

'-----[ Main ]------------------------------------------------------------

pistonPulse = 960                             ' Set piston to push plate forward
DEBUG ? pistonPulse

FOR counter = 1 TO 150                        ' for about 3.75 seconds
  GOSUB Motor_Piston_Control
NEXT

pistonPulse = 760                             ' Set piston to let go
DEBUG ? pistonPulse

DO                                            ' Piston & motor neutral indefinately
  GOSUB Motor_Piston_Control
LOOP
```

```
'-----[ Subroutine Motor_Piston_Control ]-------------------------------------

' Must call at least every 500 ms.

Motor_Piston_Control:

  PULSOUT MOTOR, motorPulse                  ' Motor control pulse
  PULSOUT PISTON, pistonPulse                ' Piston control pulse
  PAUSE 20

  RETURN

'-----[ Subroutine ST_Controller_Init ]---------------------------------------

' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on channels,
' pause antoher 1/10 s, then neutral pulses for ~ 2 s so that motor/valve
' controller gives BS2 control.

ST_Controller_Init:

  motorPulse = 760                             ' Neutral pulse durations.
  pistonPulse = 760

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  RETURN
```

Your Turn

You can set both the motorPulse and pistonPulse values before calling the **`Piston_Motor_Control`** subroutine.

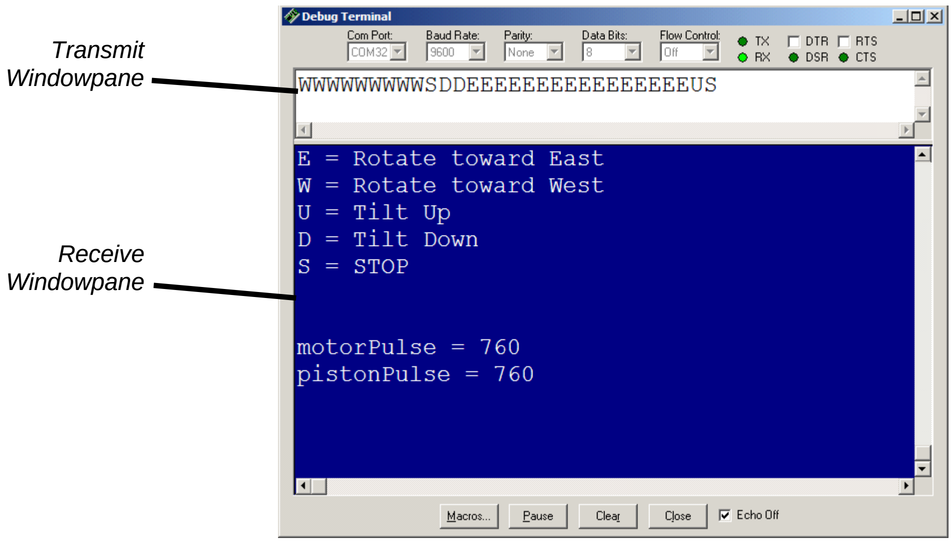
√ Write a Main routine that tips the platform forward, rotates from east to west for about 45°, then releases the pneumatic cylinder, allowing the platform to fall back, and then rotates the platform 45° from west back to east.

√ Modify the program so that it turns to face east, and tilts the platform forward so that it faces low in the sky. Each 10 seconds, it turns 1/12th of the way toward west. After 30 seconds, the platform tilts back, so that it faces high in the sky. Continue 1/12th turns every 10 seconds for 60 seconds. Then, tilt low in the sky again for the last 30 seconds.

ACTIVITY #3: TERMINAL CONTROLLED SOLAR TRACKER

For testing and adjustment, it's often helpful to be able to connect the BASIC Stamp to a PC and use the Debug Terminal to control the Solar Tracker's motions. Figure 2-3 shows an example of how the next program allows you to control the Solar Tracker.

Figure 2-3 Debug Terminal Controlling Solar Tracker



With this program, you can simply click the Debug Terminal's transmit windowpane and start typing characters from the menu to control the Solar Tracker. Each time you type W, the Solar Tracker will rotate its platform a little faster toward the west. S causes it to stop. D actuates the piston, tilting the platform forward. D is for Down. Each time you type E, the platform rotates a little faster toward the east (or in some cases slower to the west if it was already moving that direction). U is for up, which drains the air from the cylinder and allows the platform to fall back and face further up in the sky.

Same Program – Different Main Routine

Terminal Controlled Solar Tracker.bs2 uses the same variables and subroutines as the examples in the previous two activities, and the code in the Main routine is what does the work. This time, we'll try the code first, and then look at how it works.

Example Program: Terminal Controlled Solar Tracker.bs2

This example program makes it possible to control the Solar Tracker Platform by typing characters into the Debug Terminal's transmit windowpane.

√ Pump up the air tank if needed.
√ Open the manual air valve.
√ Turn on power to the Solar Tracker.
√ Load Terminal Controlled Solar Tracker.bs2 into the BASIC Stamp.
√ Keep in mind, you can stop the Solar Tracker at any time by pressing the S key on your keyboard driving it with the Debug Terminal.
√ Try typing the character sequence shown in Figure 2-3 into the Debug Terminal's transmit windowpane. The tracker platform should rotate west, tilt forward (down), rotate east, and then tilt backward (up), and then stop.

```
' Terminal Controlled Solar Tracker.bs2
' Control the Solar Tracker by typing characters into the Debug Terminal's
' transmit windowpane.

' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ Declarations ]-------------------------------------------------------

MOTOR          PIN     14                    ' Red header
PISTON         PIN     15                    ' Black header

PULSE_DELAY    CON     20                    ' 20 ms between pulses

motorPulse     VAR     Word
pistonPulse    VAR     Word
counter        VAR     Byte
char           VAR     Byte

'-----[ Initializataion ]----------------------------------------------------

GOSUB ST_Controller_Init                     ' Required, takes ~ 4 s.

' Display menu
DEBUG CLS, "E = Rotate toward East", CR, "W = Rotate toward West", CR,
           "U = Tilt Up", CR, "D = Tilt Down", CR, "S = STOP", CR, CR

'-----[ Main ]---------------------------------------------------------------

DO                                           ' DO from DO...LOOP

  SERIN 16, 84, 400, No_Serin, [char]        ' Get a character from RX pane

  SELECT char                                ' Decide what to do with char
    CASE "W", "w"                            ' W for west
      motorPulse = motorPulse - 10 MIN 500   ' Subtract 10 for each W
    CASE "E", "e"                            ' E for east
      motorPulse = motorPulse + 10 MAX 1000  ' Add 10 for each E
    CASE "S", "s"                            ' S for stop motor
      motorPulse = 760                       ' Set motor to stop
    CASE "D", "d"                            ' D for down
      pistonPulse = 960                      ' Push piston platform w/ piston
    CASE "U", "u"                            ' U for up
      pistonPulse = 760                      ' Release pressure from cylendar
    CASE ELSE                                ' If no char for 400 ms
      motorPulse = 760                       ' Stop motor
      pistonPulse = 760                      ' Release piston
    ENDSELECT                                ' End of char decision list

  No_Serin:                                  ' Timeout label
  GOSUB Motor_Piston_Control                 ' Call control sub

  ' Display motor and piston pulse values
  DEBUG CRSRXY, 0, 7, ? motorPulse, ? pistonPulse

LOOP                                         ' LOOP from DO...LOOP

'-----[ Subroutine Motor_Piston_Control ]------------------------------------

' Must call at least every 500 ms.

Motor_Piston_Control:

  PULSOUT MOTOR, motorPulse                  ' Motor control pulse
  PULSOUT PISTON, pistonPulse                ' Piston control pulse
  PAUSE 20

  RETURN

'-----[ Subroutine ST_Controller_Init ]--------------------------------------
```

```
' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on channels,
' pause antoher 1/10 s, then neutral pulses for ~ 2 s so that motor/valve
' controller gives BS2 control.

ST_Controller_Init:

  motorPulse = 760                              ' Neutral pulse durations.
  pistonPulse = 760

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  RETURN
```

How Terminal Controlled Solar Tracker.bs2 Works

The only thing different about this program is the Main routine. If you have already been through *What's a Microcontroller?* or *Robotics with the Boe-Bot*, most of that code will look familiar, with maybe one line that you'll want to look up in the *BASIC Stamp Manual.*

√ Look up and read about any commands that do not look familiar to you in the *BASIC Stamp Manual* (a free download from www.parallax.com) or look in the BASIC Stamp Editor's Help.

All of the commands in the Main routine are nested in a **DO…LOOP**. The Main routine starts with **DO** and ends with **LOOP**, which cause all the code in between to be repeated indefinitely.

```
'-----[ Main ]-----------------------------------------------------------------

DO                                             ' DO from DO...LOOP
```

If you have used the **DEBUGIN** command before, you might be wondering why the **SERIN** command was used here to get a character from the Debug Terminal's transmit windowpane. The reason is because the **SERIN** command has a ***Timeout*** argument that prevents the **SERIN** command from waiting forever if it doesn't get a character. **DEBUGIN** does not have that feature.

```
  SERIN 16, 84, 400, No_Serin, [char]        ' Get a character from RX pane
```

SERIN 16, 84, [char] would be the same as DEBUGIN char. The 16 is for pin 16. Since I/O pins only go from 0 to 15, ***16*** is a special argument that makes the **SERIN** command use the BASIC Stamp module's SIN pin for communication with the PC.

The **SERIN** command's ***Timeout*** argument is important here because the main routine needs to call the **Motor_Piston_Control** subroutine at least once every 500 ms. Note that the **SERIN** command's ***Timeout*** argument is 400, and the ***Tlabel*** argument is **No_Serin.** So, if no character is received in 400 ms, the program jumps to the **No_Serin** label (which is later in the Main routine), and continues from there, calling the **Motor_Piston_Control** subroutine within the required 500 ms.

If you type a character into the Debug Terminal's transmit windowpane, the **SERIN** command stores the character in a variable named **char**. Next, a **SELECT…CASE** statement evaluates the **char** variable on a case-by-case basis and decides what action to take. For example, if you typed either "W" or "w", the **SELECT…CASE** statement subtracts 10 from the **motorPulse** variable. If **motorPulse** started out at 760 (stopped) and you type "W" 12 times, **motorPulse** will store 640, and the platform will rotate pretty rapidly to the west. Each time you type "E" or "e", the **SELECT…CASE** statement will add 10 to **motorPulse** to make it turn the other direction. To make it stop, you can type "S", or "s", and so on… If you hit the wrong key, the **`CASE ELSE`** condition sets both the **`motorPulse`** and **`pistonPulse`** variables to neutral, 760.

```
SELECT char                                 ' Decide what to do with char
  CASE "W", "w"                             ' W for west
    motorPulse = motorPulse - 10 MIN 500    ' Subtract 10 for each W
  CASE "E", "e"                             ' E for east
    motorPulse = motorPulse + 10 MAX 1000   ' Add 10 for each E
  CASE "S", "s"                             ' S for stop motor
    motorPulse = 760                        ' Set motor to stop
  CASE "D", "d"                             ' D for down
    pistonPulse = 960                       ' Push piston platform w/ piston
  CASE "U", "u"                             ' U for up
    pistonPulse = 760                       ' Release pressure from cylendar
  CASE ELSE                                 ' If no char for 400 ms
    motorPulse = 760                        ' Stop motor
    pistonPulse = 760                       ' Release piston
  ENDSELECT                                 ' End of char decision list
```

At this point, either the **`SERIN`** command timed out and sent the program to the **`No_Serin`** label, or the **`SELECT…CASE`** statement just finished figuring out what to do. Either way, it's time to send another update to the **`Motor_Piston_Control`** subroutine.

```
No_Serin:                                   ' Timeout label
GOSUB Motor_Piston_Control                  ' Call control sub
```

This program displays the values of **`motorPulse`** and **`pistonPulse`** below the menu so that you can see what values they hold after you have pressed various keys.

```
' Display motor and piston pulse values
DEBUG CRSRXY, 0, 7, ? motorPulse, ? pistonPulse
```

When the program gets to this **`LOOP`** command, it jumps back to the **`DO`** at the beginning of the Main routine.

```
LOOP                                          ' LOOP from DO...LOOP
```

Your Turn

√ Modify the program so that the motor responds more quickly (turns faster) for each time you enter a "W" or "E".

Chapter #3: Tracking with Sensors

Now that you've got the sensing and control aspects down individually, it's time to put the two together and make the Solar Tracker track a light source. This chapter will focus on indoor experiments where your Solar Tracker can stay connected to a PC so that you can observe what the sensors sense in the Debug Terminal. After you are done with this chapter, your challenge will be to adjust the program so that it tracks reliably outside.

It's important when developing a product or application to focus on each basic component individually. After you get several components working well, then combine them and trouble-shoot as needed. In keeping with this approach, this chapter examines each aspect of using sensors to track a light source individually, and then combines them at the end, as follows:

- Experiment with tracking a light source's horizontal motion.
- Incorporate and test the limit switches to verify that they will prevent the Solar Tracker from over-rotating.
- Incorporate some constants into your code so that you can adjust the way the Solar Tracker behaves by just changing some values at the beginning of the program.
- Vertical light tracking.
- Finally, combine vertical and horizontal tracking with limit switch reaction for a full-featured light tracking mechanism.

ACTIVITY #1: HORIZONTAL TRACKING

In this activity, you will examine and test code that makes the Solar Tracker rotate and align its platform with a light source. You will also modify the code so that it does not attempt to make any adjustments if it's too dark.

Horizontal Tracking Code

This code from the next example program's Main routine makes it convenient to track a light source horizontally. The first thing it does is calculate the difference between the **`time(LIGHT_EAST)`** and **`time(LIGHT_WEST)`** measurements.

```
error = time(LIGHT_WEST) - time(LIGHT_EAST)
```

This difference has to be compared with the overall light levels; otherwise, it doesn't mean anything. For example, a difference of 700 might not be that large if the average of the two horizontal light sensors is 30,000. However, a difference of 700 is large if the average of the two light sensors is only 2,000. So, the command **`average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2`** takes the average of the two measurements, which gives a good indication of how bright it is.

```
average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2
```

Next, divide the error into the average. If the result is smaller than some threshold value (6 in this case), it means the Solar Tracker needs to start moving. If you want to make the Solar Tracker more sensitive to differences in light measurements, pick a number larger than 6. If you want it to be less sensitive, pick a smaller number.

```
IF average / ABS(error) < 6 THEN
```

Which direction does the Solar Tracker need to rotate? It depends on which of the two light measurements were larger. If the **`time(LIGHT_WEST)`** measurement is larger, it means the light source is to the west. So, the

Solar Tracker's motor needs to turn clockwise with **`motorPulse = 650`**, which makes the platform rotate toward the west. Otherwise, the Solar Tracker needs to turn east, with **`motorPulse = 870`**.

```
IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
  motorPulse = 650
ELSE
  motorPulse = 870
ENDIF
```

Now, if **`average / ABS(error)`** is greater than 6, there's no reason to rotate the Solar Tracker. So this else condition sets **`motorPulse`** to 760, which means the motor will not turn.

```
ELSE
  motorPulse = 760
ENDIF
```

Example Program: Horizontal Tracking.bs2

Horizontal Tracking is a combination of programs from previous chapters. The only new code is the calculations and **`IF…THEN`** statements in the Main routine.

- √ Examine the code in the Main routine and make sure you understand it.
- √ Use a 60 W incandescent desk lamp to test this code indoors.
- √ Keep the light direction sensor out of direct sunlight from the windows so that it will see the desk lamp.
- √ Load Horizontal Tracking.bs2 into the BASIC Stamp.
- √ Hold the desk lamp in the 1 yard or 1 meter range above the Solar Tracker's light direction sensor. If the light is directly above the direction sensor and shining straight at it, it should not move. If you move the light to the Solar Tracker's east or west, it should rotate to follow the light source.
- √ Be careful not to over-rotate the Solar Tracker.

```
' Horizontal Tracking.bs2
' Control horizontal rotation with horizontal light direction sensor.

' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ I/O Pin Definitions ]-----------------------------------------------

PISTON           PIN     15                  ' Black header
MOTOR            PIN     14                  ' Red header

LED_TOP          CON     11
LED_WEST         CON     10
LED_BOTTOM       CON     9
LED_EAST         CON     8

SWITCH_WEST       PIN     7
SWITCH_EAST       PIN     6

'-----[ Constants ]---------------------------------------------------------

LIGHT_TOP        CON     3
LIGHT_WEST       CON     2
LIGHT_BOTTOM     CON     1
LIGHT_EAST       CON     0

PULSE_DELAY      CON     20                  ' 20 ms between pulses

'-----[ Variables ]---------------------------------------------------------

motorPulse       VAR     Word
pistonPulse      VAR     Word
counter          VAR     Word
```

```
average          VAR     Word
error           VAR     Word
time             VAR     Word(4)
index            VAR     Nib
pindex           VAR     index
x                VAR     Nib
y                VAR     Nib

'-----[ Initializataion ]----------------------------------------------------

GOSUB ST_Controller_Init                     ' Required, takes ~ 4 s.
DEBUG CLS, "   Light Levels",
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"

'-----[ Main ]---------------------------------------------------------------

DO

  GOSUB Get_Decay_Times
  GOSUB Display_Decay_Times

  error = time(LIGHT_WEST) - time(LIGHT_EAST)
  average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2
  IF average / ABS(error) < 6 THEN
    IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
      motorPulse = 650
    ELSE
      motorPulse = 870
    ENDIF
  ELSE
    motorPulse = 760
  ENDIF

  GOSUB Motor_Piston_Control

LOOP

'-----[ Subroutine Get_Decay_Times ]-----------------------------------------

Get_Decay_Times:

  FOR pindex = LED_EAST TO LED_TOP
    HIGH pindex
    PAUSE 1
    RCTIME pindex, 1, time(pindex - LED_EAST)
  NEXT

  RETURN

'-----[ Subroutine Display_Decay_Times ]-------------------------------------

Display_Decay_Times:

  FOR index = LIGHT_EAST TO LIGHT_TOP
    LOOKUP index, [12, 6, 0, 6], x
    LOOKUP index, [4, 6, 4, 2], y
    DEBUG CRSRXY, x, y, DEC5 time(index), CR
  NEXT
  DEBUG CR, CR

  RETURN

'-----[ Subroutine Motor_Piston_Control ]------------------------------------

' Must call at least every 500 ms.

Motor_Piston_Control:
  PULSOUT MOTOR, motorPulse
  PULSOUT PISTON, pistonPulse
  PAUSE 20
```

```
  RETURN

'-----[ Subroutine ST_Controller_Init ]--------------------------------------

' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on channels,
' pause antoher 1/10 s, then neutral pulses for ~ 2 s so that motor/valve
' controller gives BS2 control.

ST_Controller_Init:

  motorPulse = 760                             ' Neutral pulse durations.
  pistonPulse = 760

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  RETURN
```

Your Turn

If it's cloudy outside, the light becomes more evenly distributed and difficult for the Solar Tracker to track. When you take the Solar Tracker outdoors, you'll need to adjust a value that prevents it from attempting to adjust under these circumstances. The code below has an extra **IF…THEN** statement at the beginning, and an extra **ELSE** condition at the end. Currently, it compares the **time(LIGHT_WEST)** and **time(LIGHT_EAST)** measurements to 30000, which is pretty dark. If both light measurements are less than 30,000, the code moves on to check and see if there's enough difference between the two sensors for a position adjustment. If either of the light measurements is greater than 30000, the code below skips everything, and **ELSE motorPulse = 760 ENDIF** makes the motor stay still.

```
IF time(LIGHT_WEST) < 30000 AND time(LIGHT_EAST) < 30000 THEN
    error = time(LIGHT_WEST) - time(LIGHT_EAST)
    average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2
    IF average / ABS(error) < 6 THEN
      IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
        motorPulse = 650
      ELSE
        motorPulse = 870
      ENDIF
    ELSE
      motorPulse = 760
    ENDIF
  ELSE
    motorPulse = 760
  ENDIF
```

√ Try the Solar Tracker with the modified code. Reduce the light measurement thresholds (the 30000 values) so that they cause the Solar Tracker to stop adjusting if you cast a shadow with a fairly large object between the light direction sensor and your desk lamp.

ACTIVITY #2: DOCUMENTING THE CODE

The code example in the previous activity's Main routine had a lot of numbers that needed to be adjusted, 6, 30000, and if you want to adjust the motor speeds, 650 and 870. Here are some constant declarations. Add them to your code, and replace the values in the main routine with the constant names.

```
MAX_TIME       CON     30000
PULSE_CCW      CON     870
PULSE_CW       CON     650
PULSE_STOP     CON     760
MAX_ERROR_H    CON     6
```

ACTIVITY #3: LIMIT SWITCHES FOR HORIZONTAL TRACKING

If the Solar Tracker rotates too far in either direction, it might damage the wiring, pneumatic tubing, or both. In this activity, you will mount and test limit switches that will prevent the Solar Tracker from over-rotating.

Adding IF…THEN Statements to Prevent Rotation if a Switched is Pressed

Two additional **IF…THEN** statements can be added to the Main Routine to stop the Solar Tracker if its rotating platform has bumped into a limit switch. Notice that the code that set the **motorPulse** to **PULSE_CW (650)** now has an **IF** condition that checks to make sure the switch is sending a 1 to the I/O pin. If it's not, it means it's zero instead. In that case, the **IF…THEN** statement sets the **motorPulse** variable to **PULSE_STOP** because the platform bumped into the switch.

```
...
IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
  IF SWITCH_WEST=1 THEN motorPulse=PULSE_CW ELSE motorPulse=PULSE_STOP
ELSE
  IF SWITCH_EAST=1 THEN motorPulse=PULSE_CCW ELSE motorPulse=PULSE_STOP
...
```

Example Program: Horizontal Rotation Control with Limit Switches.bs2

This program senses when the Solar Tracker's platform has rotated far enough to bump into a limit switch and stops the motor when it detects this condition.

- √ Load the modified code into the BASIC Stamp.
- √ Test the limit switches manually first. Position the 60 W desk lamp so that its light is shining at the direction sensor at an angle that causes the Solar Tracker's platform to start rotating.
- √ Press the limit switch with your hand and verify that the motor stops.

```
' Horizontal Rotation Control with Limit Switches.bs2
' Control horizontal rotation with horizontal light direction sensor.

' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ I/O Pin Definitions ]-----------------------------------------------

PISTON          PIN     15                  ' Black header
MOTOR           PIN     14                  ' Red header

LED_TOP         CON     11
LED_WEST        CON     10
LED_BOTTOM      CON     9
LED_EAST        CON     8

SWITCH_WEST      PIN     7
SWITCH_EAST      PIN     6

'-----[ Constants ]---------------------------------------------------------

LIGHT_TOP       CON     3
LIGHT_WEST      CON     2
LIGHT_BOTTOM    CON     1
LIGHT_EAST      CON     0

PULSE_DELAY     CON     20                  ' 20 ms between pulses

MAX_TIME        CON     30000
PULSE_CCW       CON     870
PULSE_CW        CON     650
PULSE_STOP      CON     760
MAX_ERROR_H     CON     6

'-----[ Variables ]---------------------------------------------------------
```

```
motorPulse       VAR     Word
pistonPulse      VAR     Word
counter          VAR     Word
average          VAR     Word
error           VAR     Word
time             VAR     Word(4)
index            VAR     Nib
pindex           VAR     index
x                VAR     Nib
y                VAR     Nib

'-----[ Initializataion ]----------------------------------------------------

GOSUB ST_Controller_Init                     ' Required, takes ~ 4 s.
DEBUG CLS, "   Light Levels",
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"

'-----[ Main ]---------------------------------------------------------------

DO

  GOSUB Get_Decay_Times
  GOSUB Display_Decay_Times

  IF time(LIGHT_WEST) < MAX_TIME AND time(LIGHT_EAST) < MAX_TIME THEN
    error = time(LIGHT_WEST) - time(LIGHT_EAST)
    average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2
    IF average / ABS(error) < MAX_ERROR_H THEN
      IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
        IF SWITCH_WEST=1 THEN motorPulse=PULSE_CW ELSE motorPulse=PULSE_STOP
      ELSE
        IF SWITCH_EAST=1 THEN motorPulse=PULSE_CCW ELSE motorPulse=PULSE_STOP
      ENDIF
    ELSE
      motorPulse = PULSE_STOP
    ENDIF
  ELSE
    motorPulse = PULSE_STOP
  ENDIF

  GOSUB Motor_Piston_Control

LOOP

'-----[ Subroutine Get_Decay_Times ]-----------------------------------------

Get_Decay_Times:

  FOR pindex = LED_EAST TO LED_TOP
    HIGH pindex
    PAUSE 1
    RCTIME pindex, 1, time(pindex - LED_EAST)
  NEXT

  RETURN

'-----[ Subroutine Display_Decay_Times ]-------------------------------------

Display_Decay_Times:

  FOR index = LIGHT_EAST TO LIGHT_TOP
    LOOKUP index, [12, 6, 0, 6], x
    LOOKUP index, [4, 6, 4, 2], y
    DEBUG CRSRXY, x, y, DEC5 time(index), CR
  NEXT
  DEBUG CR, CR

  RETURN
```

```
'-----[ Subroutine Motor_Piston_Control ]-------------------------------------

' Must call at least every 500 ms.

Motor_Piston_Control:
  PULSOUT MOTOR, motorPulse
  PULSOUT PISTON, pistonPulse
  PAUSE 20
  RETURN

'-----[ Subroutine ST_Controller_Init ]----------------------------------------

' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on channels,
' pause antoher 1/10 s, then neutral pulses for ~ 2 s so that motor/valve
' controller gives BS2 control.

ST_Controller_Init:

  motorPulse = 760                             ' Neutral pulse durations.
  pistonPulse = 760

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  RETURN
```

ACTIVITY #4: VERTICAL TRACKING

While horizontal tracking uses a DC motor and can make fine adjustments, vertical tracking features two levels of tilt controlled by a linear actuator. They can either be used for summer vs. winter, or to get a better look at the sun in the earlier and later parts of the day.

Vertical Tracking Control Code

This is the vertical tracking test code from the next example program's Main routine. It is very similar to the horizontal tracking code.

```
  IF time(LIGHT_TOP) < MAX_TIME AND time(LIGHT_BOTTOM) < MAX_TIME THEN
    error = time(LIGHT_TOP) - time(LIGHT_BOTTOM)
    average = time(LIGHT_TOP) + time(LIGHT_BOTTOM) / 2
    IF average / ABS(error) < MAX_ERROR_V THEN
      IF time(LIGHT_BOTTOM) > time(LIGHT_TOP) THEN
        pistonPulse = PISTON_PUSH
      ELSE
        pistonPulse = PISTON_RELEASE
      ENDIF
    ENDIF
  ENDIF
```

If both light measurements are less than **MAX_TIME**, the code subtracts **time(LIGHT_BOTTOM)** from **time(LIGHT_TOP)** to calculate the error, calculates the average of the light **time(LIGHT_TOP)** and **time(LIGHT_BOTTOM)** measurements, and then divides the error into the average. If the result is les than 2 (**MAX_ERROR_V**), it means the difference it pretty large, and it's time to change the tilt. If **time(LIGHT_BOTTOM)** is greater than **time(LIGHT_TOP)**, it means the light source is low on the horizon. So the **pistonPulse** variable gets set to **PISTON_PUSH (960)**. This pushes the platform forward so that it tilts toward the horizon. Otherwise, the light source is pretty high up, so **pistonPulse** gets set to **PISTON_RELEASE**, the piston lets go, and the platform faces higher above the horizon.

Example Program: Test Tilt Control.bs2

This code has the vertical control test code discussed earlier, and the horizontal control code was removed.

- √ Make sure the pneumatic reservoir is pressurized.
- √ Load the modified code into the BASIC Stamp.
- √ Point your 60 W lamp at the light direction sensor directly above it (so that the light goes straight into the PVC pipe from 1 yard or 1 meter above). The piston should remain in its neutral position.
- √ Gradually bring the lamp toward the horizon. (Make sure the bulb is pointed directly at the light direction sensor the whole time.) At some point, the platform should abruptly tilt forward.
- √ Gradually bring the lamp back up to the positing you started it with, and the platform should tip back to its neutral position.

```
' Test Tilt Control.bs2
' {$STAMP BS2}
' {$PBASIC 2.5}

'-----[ I/O Pin Definitions ]-------------------------------------------------

PISTON           PIN     15                  ' Black header
MOTOR            PIN     14                  ' Red header

LED_TOP          CON     11
LED_WEST         CON     10
LED_BOTTOM       CON     9
LED_EAST         CON     8

SWITCH_WEST       PIN     7
SWITCH_EAST       PIN     6

'-----[ Constants ]-----------------------------------------------------------

LIGHT_TOP        CON     3
LIGHT_WEST       CON     2
LIGHT_BOTTOM     CON     1
LIGHT_EAST       CON     0

PULSE_DELAY      CON     20                  ' 20 ms between pulses

MAX_TIME         CON     30000
PULSE_CCW        CON     870
PULSE_CW         CON     650
PULSE_STOP       CON     760
PISTON_PUSH      CON     960
PISTON_RELEASE   CON     760
MAX_ERROR_H      CON     6
MAX_ERROR_V      CON     2

'-----[ Variables ]-----------------------------------------------------------

motorPulse       VAR     Word
pistonPulse      VAR     Word
counter          VAR     Word
average          VAR     Word
error           VAR     Word
time             VAR     Word(4)
index            VAR     Nib
pindex           VAR     index
x                VAR     Nib
y                VAR     Nib

'-----[ Initializataion ]-----------------------------------------------------

GOSUB ST_Controller_Init                     ' Required, takes ~ 4 s.
DEBUG CLS, "   Light Levels",
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"
```

```
'-----[ Main ]----------------------------------------------------------------

DO

  GOSUB Get_Decay_Times
  GOSUB Display_Decay_Times

  IF time(LIGHT_TOP) < MAX_TIME AND time(LIGHT_BOTTOM) < MAX_TIME THEN
    error = time(LIGHT_TOP) - time(LIGHT_BOTTOM)
    average = time(LIGHT_TOP) + time(LIGHT_BOTTOM) / 2
    IF average / ABS(error) < MAX_ERROR_V THEN
      IF time(LIGHT_BOTTOM) > time(LIGHT_TOP) THEN
        pistonPulse = PISTON_PUSH
      ELSE
        pistonPulse = PISTON_RELEASE
      ENDIF
    ENDIF
  ENDIF

  GOSUB Motor_Piston_Control

LOOP

'-----[ Subroutine Get_Decay_Times ]-----------------------------------------

Get_Decay_Times:

  FOR pindex = LED_EAST TO LED_TOP
    HIGH pindex
    PAUSE 1
    RCTIME pindex, 1, time(pindex - LED_EAST)
  NEXT

  RETURN

'-----[ Subroutine Display_Decay_Times ]-------------------------------------

Display_Decay_Times:

  FOR index = LIGHT_EAST TO LIGHT_TOP
    LOOKUP index, [12, 6, 0, 6], x
    LOOKUP index, [4, 6, 4, 2], y
    DEBUG CRSRXY, x, y, DEC5 time(index), CR
  NEXT
  DEBUG CR, CR

  RETURN

'-----[ Subroutine Motor_Piston_Control ]-------------------------------------

' Must call at least every 500 ms.

Motor_Piston_Control:
  PULSOUT MOTOR, motorPulse
  PULSOUT PISTON, pistonPulse
  PAUSE 20
  RETURN

'-----[ Subroutine ST_Controller_Init ]--------------------------------------

' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on channels,
' pause antoher 1/10 s, then neutral pulses for ~ 2 s so that motor/valve
' controller gives BS2 control.

ST_Controller_Init:

  motorPulse = 760                             ' Neutral pulse durations.
  pistonPulse = 760
```

```
FOR counter = 0 TO 199
  IF counter//100 = 0 THEN PAUSE 100
  GOSUB Motor_Piston_Control
  IF counter//50 = 0 THEN DEBUG "Initializing...", CR
NEXT

RETURN
```

Your Turn – Testing both Horizontal and Vertical

Try adding the horizontal control code to the main routine either immediately before or immediately after the vertical control code. It should provide both horizontal and vertical light tracking.

ACTIVITY #5: ALL TOGETHER NOW

This activity features an example program that tracks an indoor desk lamp (60 W in otherwise low light conditions is recommended). It can be adjusted for outdoor sunlight tracking as well. Adjustments to the user-defined constants will help the Solar Tracker reliably track. For day to day operation, the program will also need some additional modifications. These are discussed in the Your Turn section.

Tracker_Control Subroutine

The next example program has both the horizontal and vertical decision making code nested in a subroutine named **Tracker_Control**. The Main routine has to call this subroutine after it calls **Get_Decay_Times** so that it has the latest measurements. Then, **Tracker_Control** examines the light measurements and sets the **motorPulse** and **pistionPulse** variables accordingly. After that, the main routine can call the **Motor_Piston_Control** subroutine which uses those two variables to control the motor and piston.

Tuning the User Defined Constants

There are five **CON** declarations below the User Defined Constants comment in the Constants Section. They can be adjusted to change the way the Solar Tracker behaves.

```
' User Defined Constants
```

PULSE_CCW and **PULSE_CW** control how fast the motor turns when adjusting horizontal position. Numbers further from 760 will make the motor turn faster, but they should stay in the range of 500 to 1000. It's better to make the motor adjust more slowly because the IDS II controller takes time to ramp up and back down.

```
PULSE_CCW      CON     870             ' Counterclockwise motor speed
PULSE_CW       CON     650             ' Clockwise motor speed
```

If you select a high speed, the platform might overshoot its target and decide it needs to turn back. When it turns back, it might turn too far too if the speed are high, and this can lead to the platform oscillating back and forth.

MAX_TIME is a sensor measurement that indicates that it's too dark and not worth making a correction. This number should probably be much smaller for outdoor use. It can be used to prevent the Solar Tracker from getting confused by clouds and twilight. Take measurements when clouds pass and during sunset to determine what the threshold should be.

```
MAX_TIME       CON     30000           ' "Too dark" threshold
```

MAX_ERROR_H is a value that the program uses to decide if the difference between east and west light sensor measurements is large enough to make it worth adjusting the Solar Tracker's platform position. The average of the two light sensors divided by the difference between the two has to be smaller than **MAX_ERROR_H** for a platform position adjustment to be made. Larger values of **MAX_ERROR_H** make it more sensitive, smaller values make it less sensitive.

```
MAX_ERROR_H    CON     6               ' Horizontal error threshold
```

MAX_ERROR_V works the same way for the vertical platform tilt. Note that a small number is used because the change in the platform tilt is pretty large. Since the adjustment is large, the value needs to be small (large difference in up vs. down sensor) before it makes the adjustment.

```
MAX_ERROR_V    CON     2               ' Vertical error threshold
```

Example Program: Track Indoor Lamp.bs2

This example program does a pretty good job of tracking a desk lamp. If you take it outdoors, you might notice some erratic behavior. As mentioned earlier, if a cloud passes in frond of the sun, the Solar Tracker will start attempting to adjust back and forth to find the light source. Most of this can be remedied by adjusting the constants in the User Defined Constants section, but that's going to be your job.

√ Run the program, and verify its functionality indoors with a 60 W desk lamp.
√ If you have a laptop, take the Solar Tracker outside and observe the sensor measurement values while the Solar Tracker operates. You will need to make adjustments to get it to reliably adjust under full sunlight conditions.
√ You will also need to modify the program to make it track sunlight from day to day. See the Your Turn section that follows this example program for more information.

```
' Track Indoor Lamp.bs2
' Test light tracking control with an indoor lamp.

' {$STAMP BS2}                           ' Select BASIC Stamp 2 as target
' {$PBASIC 2.5}                          ' Use PBASIC 2.5 language

'-----[ I/O Pin Definitions ]----------------------------------------------

PISTON         PIN     15                ' Black header
MOTOR          PIN     14                ' Red header

LED_TOP        CON     11                ' Top LED light sensor circuit
LED_WEST       CON     10                ' West LED light sensor circuit
LED_BOTTOM     CON     9                 ' Bottom LED light sensor circuit
LED_EAST       CON     8                 ' East LED light sensor circuit

SWITCH_WEST    PIN     7                 ' West limit switch
SWITCH_EAST    PIN     6                 ' East limit switch

'-----[ Constants ]--------------------------------------------------------

' User defined constants

PULSE_CCW      CON     870               ' Counterclockwise motor speed
PULSE_CW       CON     650               ' Clockwise motor speed
MAX_TIME       CON     30000             ' "Too dark" threshold
MAX_ERROR_H    CON     6                 ' Horizontal error threshold
MAX_ERROR_V    CON     2                 ' Vertical error threshold

' Other controller constants

PULSE_STOP     CON     760               ' Stop motor signal
PISTON_PUSH    CON     960               ' Push piston signal
PISTON_RELEASE CON     760               ' Release piston signal
PULSE_DELAY    CON     20                ' 20 ms minimum between pulses

' Program constants

LIGHT_TOP        CON     3               ' Top light measurement index
```

```
LIGHT_WEST       CON     2                   ' West light measurement index
LIGHT_BOTTOM     CON     1                   ' Bottom light measurement index
LIGHT_EAST       CON     0                   ' East light measurement index

'-----[ Variables ]----------------------------------------------------------

motorPulse     VAR     Word                  ' Motor control pulse duration
pistonPulse    VAR     Word                  ' Piston control pulse duration
counter        VAR     Word                  ' Loop counter
average        VAR     Word                  ' Average measurement
error          VAR     Word                  ' Difference btwn measurements
time           VAR     Word(4)               ' Light sensor array variables
index          VAR     Nib                   ' Indexing variable
pindex         VAR     index                 ' Pin index variable alias
x              VAR     Nib                   ' x cursor position for Debug
y              VAR     Nib                   ' y cursor position for Debug

'-----[ Initializataion ]----------------------------------------------------

GOSUB ST_Controller_Init                     ' Required, takes ~ 4 s.

DEBUG CLS, "   Light Levels",                ' Display legend for measurements
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"

'-----[ Main ]---------------------------------------------------------------

DO                                           ' Repeat indefinitely (DO...LOOP)

  GOSUB Get_Decay_Times                      ' Get light sensor measurements
  GOSUB Display_Decay_Times                  ' Display measurements in Debug
  GOSUB Tracker_Control                      ' Decide what to do
  GOSUB Motor_Piston_Control                 ' Control motor and piston

LOOP

'-----[ Subroutine Get_Decay_Times ]-----------------------------------------

Get_Decay_Times:                             ' Get light measurements

  FOR pindex = LED_EAST TO LED_TOP           ' Loop through four sensors
    HIGH pindex                              ' Charge diode junction
    PAUSE 1                                  ' Wait 1 ms
    RCTIME pindex, 1, time(pindex - LED_EAST)' Time diode junction cap decay
  NEXT                                       ' Repeat FOR...NEXT loop

  RETURN                                     ' Return from subroutine

'-----[ Subroutine Display_Decay_Times ]-------------------------------------

Display_Decay_Times:                         ' Light sensor display subroutine

  FOR index = LIGHT_EAST TO LIGHT_TOP        ' Loop through the 4 measurements
    LOOKUP index, [12, 6, 0, 6], x           ' Set cursor-X position
    LOOKUP index, [4, 6, 4, 2], y            ' Set cursor-Y position
    DEBUG CRSRXY, x, y, DEC5 time(index), CR ' Display indexed measurement
  NEXT                                       ' Repeat the loop

  RETURN                                     ' Return from subroutine

'-----[ Subroutine Tracker_Control ]-----------------------------------------

' Set control variables based on sensor measurements.

Tracker_Control:

  ' East/West rotation
  ' If it's light enough, divide difference between axis light measurements
  ' into average light for the axis.  If the result is less than MAX_ERROR_H,
  ' tracker rotates platform to catch up with light source provided the limit
  ' switch for that direction is not pressed.
```

```
  IF time(LIGHT_WEST) < MAX_TIME AND time(LIGHT_EAST) < MAX_TIME THEN
    error = time(LIGHT_WEST) - time(LIGHT_EAST)
    average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2
    IF average / ABS(error) < MAX_ERROR_H THEN
      IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
        IF SWITCH_WEST=1 THEN motorPulse=PULSE_CW ELSE motorPulse=PULSE_STOP
      ELSE
        IF SWITCH_EAST=1 THEN motorPulse=PULSE_CCW ELSE motorPulse=PULSE_STOP
      ENDIF
    ELSE
      motorPulse = PULSE_STOP
    ENDIF
  ELSE
    motorPulse = PULSE_STOP
  ENDIF

  ' Up/down adjustment
  ' If it’s light enough, divide difference between axis light measurements
  ' into average light for the axis.  If the result is less than MAX_ERROR_V,
  ' tracker tilts platform to face the light source.

  IF time(LIGHT_TOP) < MAX_TIME AND time(LIGHT_BOTTOM) < MAX_TIME THEN
    error = time(LIGHT_TOP) - time(LIGHT_BOTTOM)
    average = time(LIGHT_TOP) + time(LIGHT_BOTTOM) / 2
    IF average / ABS(error) < MAX_ERROR_V THEN
      IF time(LIGHT_BOTTOM) > time(LIGHT_TOP) THEN
        pistonPulse = PISTON_PUSH
      ELSE
        pistonPulse = PISTON_RELEASE
      ENDIF
    ENDIF
  ENDIF

  RETURN

'-----[ Subroutine Motor_Piston_Control ]------------------------------------

' Send pulses similar to standard hobby servo control signals to control the
' Solar Tracker’s motor and piston.
' Must call at least every 500 ms.

Motor_Piston_Control:                        ' Subroutine label

  PULSOUT MOTOR, motorPulse                  ' Send motor control pulse
  PULSOUT PISTON, pistonPulse                ' Send piston control pulse
  PAUSE PULSE_DELAY                          ' Delay for at least PAUSE_DELAY

  RETURN                                     ' Return from subroutine

'-----[ Subroutine ST_Controller_Init ]--------------------------------------

ST_Controller_Init:                          ' Initialize IDS II controller

  motorPulse = 760                           ' Neutral pulse durations.
  pistonPulse = 760

  ' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on
  ' channels, pause antoher 1/10 s, then neutral pulses for ~ 2 s so that
  ' motor/valve controller gives BS2 control.

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  RETURN                                     ' Return from subroutine
```

Your Turn – Day to Day Operation

Aside from the tuning for outdoor operation discussed earlier, this program needs some additional modifications for day to day solar tracking. First, it needs to be able to get a rough idea of when the day has begun and when it's over. A couple of counting variables can be used for this purpose.

√ Add a command that adds 1 to the **`counter`** variable each time through the Main routine.
√ Test how long it takes to get to 500, and then calculate the time per loop repetition.

You can use a pair of counters to count an entire day if one counter variable that can go up to 65535 isn't enough. The code can watch one counter, and each time it gets to a value, it can add one to a second counter and clears the first. When the program detects that the counter is getting close to the end of the day, it can monitor the light levels. When the light levels indicate that the day is over, the code should make the Solar Tracker turn back to face east. The counter can also count the approximate time a night takes so that it waits until morning to start looking for bright light conditions.

You can prototype this in a room with a lamp. Guide the lamp over an emulated solar path over the course of a minute, and then turn the lamp off for a minute. The Solar Tracker should return to facing east, and then wait half a minute before it starts looking for light again. The difference between indoors and outdoors is that the counters will have to count to much higher values, and the User Defined Constants will have to be adjusted for brighter light conditions.

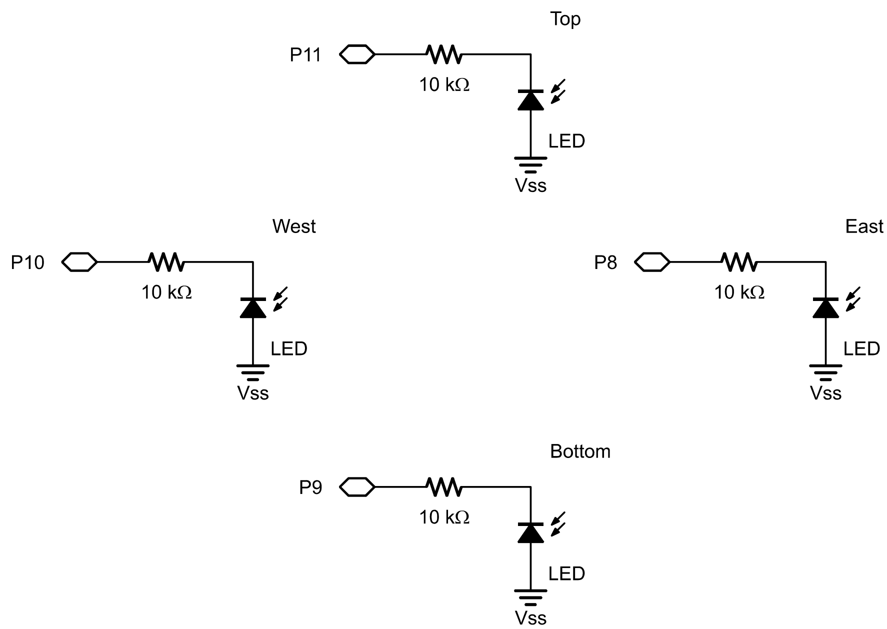
√ This is your project, good luck!

Appendix A: Indoor Light Tracking App Summary

PARTS LIST

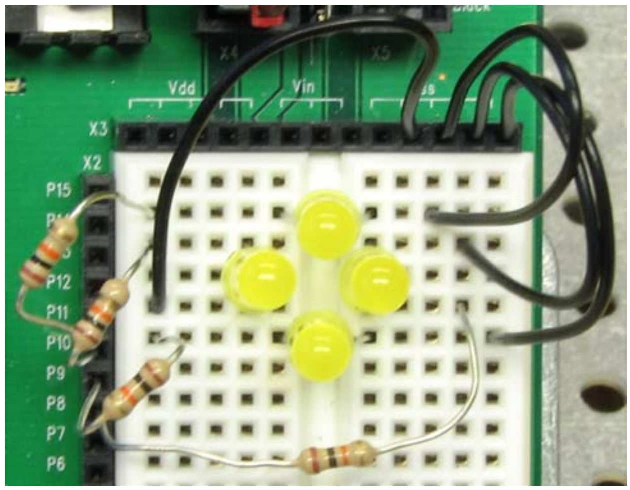
(4) LEDs – Yellow
(6) Resistors – 10 kΩ (brown-black-orange)
(2) Limit switches (normally open)
(2) Resistors – 200 Ω (red-red-brown)
(1) PVC pipe, ¾" long X ½" diameter
(misc) jumper wires
(1) Desk lamp with 60 W incandescent bulb

SCHEMATIC – LIGHT DIRECTION SENSOR AND LIMIT SWITCH



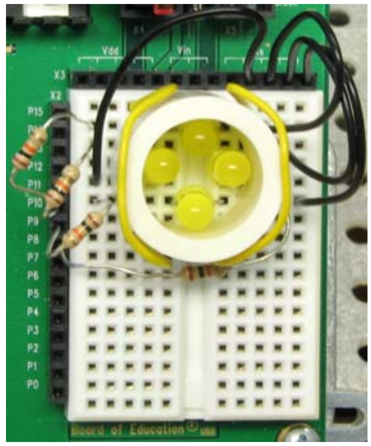
RECOMMENDED WIRING

Keep in mind that the LED cathodes (pin closest to flat spot on bottom edge of plastic case) get connected to the resistors, and the anodes get connected to ground (Vss). Make sure to trim the LED leads so that the underside of the LED cases are pretty close to flush with the top of the breadboard. Leave about 1/8 " extra for wiggle room.

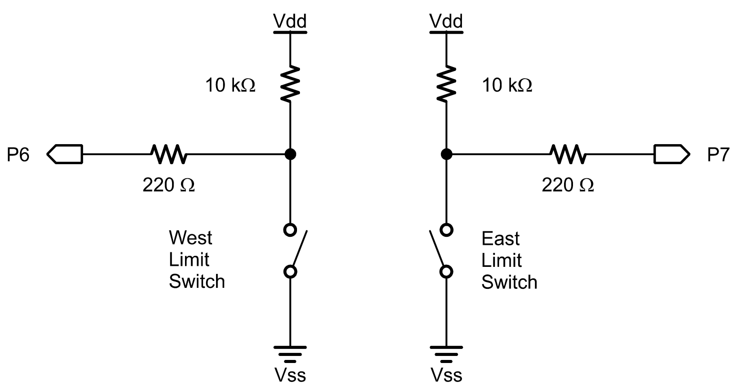


FINAL DIRECTION SENSOR

Tops of LEDs should be about 3/8” below the top of the PVC tube. Picture shows jumper wires holding the pipe in place



SCHEMATIC – LIMIT SWITCHES



SOURCE CODE

```
' Track Indoor Lamp.bs2
' Test light tracking control with an indoor lamp.

' {$STAMP BS2}                                 ' Select BASIC Stamp 2 as target
' {$PBASIC 2.5}                                ' Use PBASIC 2.5 language

'-----[ I/O Pin Definitions ]----------------------------------------------

PISTON         PIN     15                      ' Black header
MOTOR          PIN     14                      ' Red header

LED_TOP        CON     11                      ' Top LED light sensor circuit
LED_WEST       CON     10                      ' West LED light sensor circuit
LED_BOTTOM     CON     9                       ' Bottom LED light sensor circuit
LED_EAST       CON     8                       ' East LED light sensor circuit

SWITCH_WEST    PIN     7                       ' West limit switch
SWITCH_EAST    PIN     6                       ' East limit switch

'-----[ Constants ]--------------------------------------------------------

' User defined constants

PULSE_CCW      CON     870                     ' Counterclockwise motor speed
PULSE_CW       CON     650                     ' Clockwise motor speed
MAX_TIME       CON     30000                   ' "Too dark" threshold
MAX_ERROR_H    CON     6                       ' Horizontal error threshold
MAX_ERROR_V    CON     2                       ' Vertical error threshold

' Other controller constants

PULSE_STOP     CON     760                     ' Stop motor signal
PISTON_PUSH    CON     960                     ' Push piston signal
PISTON_RELEASE CON     760                     ' Release piston signal
PULSE_DELAY    CON     20                      ' 20 ms minimum between pulses

' Program constants

LIGHT_TOP        CON     3                     ' Top light measurement index
LIGHT_WEST       CON     2                     ' West light measurement index
LIGHT_BOTTOM     CON     1                     ' Bottom light measurement index
LIGHT_EAST       CON     0                     ' East light measurement index

'-----[ Variables ]--------------------------------------------------------

motorPulse     VAR     Word                    ' Motor control pulse duration
pistonPulse    VAR     Word                    ' Piston control pulse duration
counter        VAR     Word                    ' Loop counter
average        VAR     Word                    ' Average measurement
error          VAR     Word                    ' Difference btwn measurements
time           VAR     Word(4)                 ' Light sensor array variables
index          VAR     Nib                     ' Indexing variable
pindex         VAR     index                   ' Pin index variable alias
x              VAR     Nib                     ' x cursor position for Debug
y              VAR     Nib                     ' y cursor position for Debug

'-----[ Initializataion ]--------------------------------------------------

GOSUB ST_Controller_Init                       ' Required, takes ~ 4 s.

DEBUG CLS, "   Light Levels",                  ' Display legend for measurements
      CRSRXY, 8, 3, "U", CRSRXY, 8, 5, "D",
      CRSRXY, 6, 4, "W", CRSRXY, 10, 4, "E"

'-----[ Main ]-------------------------------------------------------------

DO                                             ' Repeat indefinitely (DO...LOOP)
```

```
  GOSUB Get_Decay_Times                       ' Get light sensor measurements
  GOSUB Display_Decay_Times                   ' Display measurements in Debug
  GOSUB Tracker_Control                       ' Decide what to do
  GOSUB Motor_Piston_Control                  ' Control motor and piston

LOOP

'-----[ Subroutine Get_Decay_Times ]-----------------------------------------

Get_Decay_Times:                              ' Get light measurements

  FOR pindex = LED_EAST TO LED_TOP            ' Loop through four sensors
    HIGH pindex                               ' Charge diode junction
    PAUSE 1                                   ' Wait 1 ms
    RCTIME pindex, 1, time(pindex - LED_EAST)' Time diode junction cap decay
  NEXT                                        ' Repeat FOR...NEXT loop

  RETURN                                      ' Return from subroutine

'-----[ Subroutine Display_Decay_Times ]-------------------------------------

Display_Decay_Times:                          ' Light sensor display subroutine

  FOR index = LIGHT_EAST TO LIGHT_TOP         ' Loop through the 4 measurements
    LOOKUP index, [12, 6, 0, 6], x            ' Set cursor-X position
    LOOKUP index, [4, 6, 4, 2], y             ' Set cursor-Y position
    DEBUG CRSRXY, x, y, DEC5 time(index), CR ' Display indexed measurement
  NEXT                                        ' Repeat the loop

  RETURN                                      ' Return from subroutine

'-----[ Subroutine Tracker_Control ]-----------------------------------------

' Set control variables based on sensor measurements.

Tracker_Control:

  ' East/West rotation
  ' If it's light enough, divide difference between axis light measurements
  ' into average light for the axis.  If the result is less than MAX_ERROR_H,
  ' tracker rotates platform to catch up with light source provided the limit
  ' switch for that direction is not pressed.

  IF time(LIGHT_WEST) < MAX_TIME AND time(LIGHT_EAST) < MAX_TIME THEN
    error = time(LIGHT_WEST) - time(LIGHT_EAST)
    average = time(LIGHT_EAST) + time(LIGHT_WEST) / 2
    IF average / ABS(error) < MAX_ERROR_H THEN
      IF time(LIGHT_WEST) > time(LIGHT_EAST) THEN
        IF SWITCH_WEST=1 THEN motorPulse=PULSE_CW ELSE motorPulse=PULSE_STOP
      ELSE
        IF SWITCH_EAST=1 THEN motorPulse=PULSE_CCW ELSE motorPulse=PULSE_STOP
      ENDIF
    ELSE
      motorPulse = PULSE_STOP
    ENDIF
  ELSE
    motorPulse = PULSE_STOP
  ENDIF

  ' Up/down adjustment
  ' If it's light enough, divide difference between axis light measurements
  ' into average light for the axis.  If the result is less than MAX_ERROR_V,
  ' tracker tilts platform to face the light source.

  IF time(LIGHT_TOP) < MAX_TIME AND time(LIGHT_BOTTOM) < MAX_TIME THEN
    error = time(LIGHT_TOP) - time(LIGHT_BOTTOM)
    average = time(LIGHT_TOP) + time(LIGHT_BOTTOM) / 2
    IF average / ABS(error) < MAX_ERROR_V THEN
      IF time(LIGHT_BOTTOM) > time(LIGHT_TOP) THEN
        pistonPulse = PISTON_PUSH
      ELSE
```

```
        pistonPulse = PISTON_RELEASE
      ENDIF
    ENDIF
  ENDIF

  RETURN

'-----[ Subroutine Motor_Piston_Control ]-------------------------------------

' Send pulses similar to standard hobby servo control signals to control the
' Solar Tracker's motor and piston.
' Must call at least every 500 ms.

Motor_Piston_Control:                        ' Subroutine label

  PULSOUT MOTOR, motorPulse                  ' Send motor control pulse
  PULSOUT PISTON, pistonPulse                ' Send piston control pulse
  PAUSE PULSE_DELAY                          ' Delay for at least PAUSE_DELAY

  RETURN                                     ' Return from subroutine

'-----[ Subroutine ST_Controller_Init ]---------------------------------------

ST_Controller_Init:                          ' Initialize IDS II controller

  motorPulse = 760                           ' Neutral pulse durations.
  pistonPulse = 760

  ' Pause 1/10 s, 100 neutral pulses for ~ 2 s to indicate activity on
  ' channels, pause antoher 1/10 s, then neutral pulses for ~ 2 s so that
  ' motor/valve controller gives BS2 control.

  FOR counter = 0 TO 199
    IF counter//100 = 0 THEN PAUSE 100
    GOSUB Motor_Piston_Control
    IF counter//50 = 0 THEN DEBUG "Initializing...", CR
  NEXT

  RETURN                                     ' Return from subroutine
```