Mechatronics and Pneumatics Kit Manual
Introduction

This document describes the components of the mechatronics and pneumatics kit issued to students of ME2110 for use in the final project for the course. The main controller box uses a Basic Stamp BS2p40 to interface with various sensors and actuators via Atmel AVR microcontrollers. The Basic Stamp is programmed serially over a RS-232 cable from a PC using the PBASIC programming language. There are five electromechanical actuators provided with the kit: two DC motors, one stepper motor, a large solenoid, and a small solenoid. In addition, a pneumatic system that contains two one-way pneumatic actuators is included. They are controlled via solenoid valves. There are four sensors included with the kit: two switches, one IR distance sensor, and one encoder. The main components of the kit are identified in Interactive 1.

Included in this document are explanations of the controller box, general PBASIC programming, descriptions of each item of the kit, and how to program for each item. The appearance of items may change, but functionality and programming match that in this manual.
INTERACTIVE 1: Components of the ME2110 Mechatronics and Pneumatics Kit
(iBooks Version is necessary for interactive features.)
SECTION 1

Conventions in this text

This text will feature numerous code samples. PBASIC commands that should be typed as is are set in a fixed-width font. In addition, every attempt has been made to match the default syntax highlighting format in the Parallax PBASIC programming environment. This means that comments are set in green, PBASIC commands in blue, and strings printing to the DEBUG window in red. Command names within the text are not colored.

Variable names that are explicitly defined in a given code snippet are also set as fixed-width, typically in lower case. Variable that have not been explicitly defined or are representative of parameters that the user must define are in italics.

Where user code can be inserted is also indicated by italics.

An example code snippet is shown below to demonstrate these conventions.

```
'use the auxiliary I/O pins
AUXIO

'begin timer
SEROUT 10, 247, [0, config, counter]

'continually check the value of pin 11
check:

'when pin 11 is high,
'jump to the time_up label
IF IN11 = 1 THEN time_up

GOTO check

time_up:

DEBUG "Timer is finished", CR

program code to run once timer is finished
```
The controller box that is included with the kit, shown in Figure 1, is based around a Parallax Basic Stamp BS2p40, which sends serial commands to various Atmel AVR microcontrollers. The AVR microcontrollers directly interact with the various sensors and actuators included in the kit.
A picture of the box with the cover removed is shown in Figure 2. In this picture, the BASIC Stamp BS2p40 and the various AVR microcontrollers are visible. The cover of the box has been removed in the picture for illustration purposes only. It should not be removed during operation.

The various input and output ports are shown in Figure 3. The connectors, shown in Figure 4, are sized to help ensure that components are connected to the proper ports. Notice that one side of the box contains only outputs (the left side in Figure 3), while the other contains the inputs (the right side in Figure 3). Also note that the box is labeled differently than it should be programmed. The translation between the labels on the box and what should be programmed is shown in Table 1.
The box is programmed in PBASIC using a serial, RS-232 connection to a computer. Many modern computers no longer have RS-232 ports. However, there are USB-to-serial converters that are suitable for programming. The details of programming are discussed in the next section.

**Connecting to the Connectors**

1. Strip 1/4” of insulation from the wires to be connected.
2. Place the wires in the correct port of the connector.
3. Tighten the screws on the top of the connector.
4. Be sure to clamp on the conducting portion of the wire.

---

**Table 1: Box Label to Program Translation**

<table>
<thead>
<tr>
<th>INPUTS AND SENSORS</th>
<th>OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOX LABEL</td>
<td>PROGRAM</td>
</tr>
<tr>
<td>START</td>
<td>IN0</td>
</tr>
<tr>
<td>IN1</td>
<td>IN4</td>
</tr>
<tr>
<td>IN2</td>
<td>IN5</td>
</tr>
<tr>
<td>IN3</td>
<td>IN6</td>
</tr>
<tr>
<td>FLEX</td>
<td>3 (out) / 2 (in)</td>
</tr>
<tr>
<td>IR</td>
<td>AUXIO 14 (out) / 15 (in)</td>
</tr>
<tr>
<td>ENCODER</td>
<td>AUXIO 12 (out) / 13 (in)</td>
</tr>
</tbody>
</table>
Chapter 3

Programming in PBASIC

The program used to program the Basic Stamp on the controller box is called the BASIC Stamp Programming Editor. It is available on all of the desktop computers in the ME2110 studio. It is also available as a free download from Parallax. Only basics of the PBASIC language are covered here. For more detail, see the full PBASIC manual, available at the Parallax website.
Program Directives and Download

In order to properly program the ME2110 controller box, the type of chip must be specified in the programming editor. To do this, the line:

\['${STAMP \text{ BS2p}}\]

can be manually added to the first line of the program. This line is automatically added to the first line of any program saved with a .bsp extension. This line can also be added automatically by selecting BS2p from the Directive... Stamp menu item.

Additionally, some of the commands outlined in this manual require the latest version of the PBASIC language, PBASIC 2.5. An additional directive indicates that the program code should be compiled as this version is needed. To add this directive, the line:

\['${PBASIC \text{ 2.5}}\]

can be manually added to the beginning of the program. Also, the line can be added automatically by selecting Version 2.5 from the Directive... PBASIC menu item.

At the beginning of each program, also place a \texttt{PAUSE 1000} (see Section 2.2.4 for explanation of the PAUSE command). This allows time for the controller box to properly boot before running any other commands.

Also, most of the components in the controller box are controlled via serial communication. This communication takes the BASIC stamp some time to do. As a result, it is recommended that a short PAUSE command be placed immediately following any SEROUT command. The recommended value is \texttt{PAUSE 100}.

To download the program from the computer to the controller box, use the \texttt{▶}-shaped button on the interface. The program can also be downloaded via the Run... Run menu item. Completing either action will overwrite the existing code on the controller box and immediately begin execution of the new code.
## Section 2

### General PBASIC Programming

#### 2.2.1 PIN AND CON

The commands CON and PIN allow names to be assigned to constants and pin numbers, respectively, and should be issued at the beginning of the program. These commands both allow the program to be more easily understood by a third party, as constants and/or pins that are referenced repeatedly can be given descriptive names. For example, if pin 14 is assigned to a DC motor, then the command:

```
DC_motor PIN 14
```

would assign the name `DC_motor` to pin 14, allowing commands for the DC motor to include the descriptive name, `DC_motor`, rather than just the number 14, making interpretation of the code easier.

Similarly, the `CON` command allows a descriptive name to be assigned to a constant value, indicating its meaning and making interpretation of the code simpler. For example, if the number 24 was used repeatedly in the code as a flag condition then:

```
flag CON 24
```

would allow the name `flag` to be used to indicate its function.

#### 2.2.2 VAR

In PBASIC, all variables and their sizes must be declared at the beginning of the program. The possible sizes of variables are shown in Table 2.

---

**THIS SECTION WILL COVER**

1. The declaration of variables, pins and constants
2. Controlling program flow with:
   a. PAUSE
   b. DEBUG
   c. Labels and GOTO
   d. GOSUB... RETURN
3. Looping methods in PBASIC such as:
   a. For... Next
   b. If... Then
   c. Do... Loop
4. MAINIO and AUXIO
Variables are defined using the following format:

`variable_name VAR type`

If a variable is assigned a number outside the allowable range for its size, the editor will not return an error, but the program will behave erratically. Some examples of defining variables are provided below.

`'counter between 0 and 255`
`counter VAR byte`

`'switch can be either 0 or 1`
`switch VAR bit`

### 2.2.3 HIGH, LOW, AND TOGGLE

The I/O pins on the BASIC Stamp can be used as digital outputs using the HIGH and LOW commands. Issuing the HIGH command sets the specified pin number to five volts. The LOW commands sets the specified pin number to zero volts. The TOGGLE command switches the state of the specified pin; sending it high if it was low and sending it low if it was high. An example demonstrating the syntax of the HIGH, LOW, and TOGGLE commands is provided below.

`'set pin 11 to five volts`
`HIGH 11`

`'set pin 12 to zero volts`
`LOW 12`

`'set pin 11 low, as it was previously high`
`TOGGLE 11`

`'set pin 12 high, as it was previously low`
`TOGGLE 12`

### 2.2.4 PAUSE

The PAUSE command suspends the execution of a program a specified amount of time. This command can be useful in controlling the timing of a program. It does not deactivate the box, but simply pauses the controller from executing any code for a given time interval. For example, if a pin is set high and the program encounters a five-second PAUSE command, the pin will remain high for the five-second duration of the pause.

The syntax for the PAUSE command is as follows:

`PAUSE delay_time`

### TABLE 2: Variable Types and Sizes

<table>
<thead>
<tr>
<th>VAR TYPE</th>
<th>NUMBER RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>0-1</td>
</tr>
<tr>
<td>NIB</td>
<td>0-7</td>
</tr>
<tr>
<td>BYTE</td>
<td>0-255</td>
</tr>
<tr>
<td>WORD</td>
<td>0-65535</td>
</tr>
</tbody>
</table>
In this command line, the *delay_time* is the amount of time (in milliseconds) the program should be paused in its current state.

An example of a program using the PAUSE command is provided below.

```plaintext
HIGH 11 'set pin 11 high
PAUSE 5000 'pause 5000 milliseconds
LOW 11 'set pin 11 low
```

**KEY POINT:** Note that after SEROUT commands that have timing built into them, the BASIC Stamp no longer has knowledge of the timing of the recipient of the SEROUT command. For example, if a SEROUT command is issued to run a motor for two seconds, the Stamp will run the command following the SEROUT declaration as soon as the serial communication is completed; it will not wait the two seconds the motor is supposed to run before issuing the next command.

### 2.2.5 DEBUG

The DEBUG command is used solely for the purpose of observing the execution of a program; it is not able to change values of variables or affect the flow of a program. It can be helpful in finding the bugs in a program or ensuring that the program is working as planned. The debug window can be opened several ways. In the Basic Stamp editor, there is a button on the tool-bar that looks like an electronic chip covered by a magnifying glass. If you click on this icon, the debug window will open. The debug window can also be opened by selecting *Debug... New* from the *Run* menu. Most often, the debug window will be opened automatically when the program encounters a DEBUG command during runtime while connected to a computer.

The **DEBUG** command controls the output to the debug window. If no DEBUG commands are issued within the program, nothing will be output to the debug window. The DEBUG command can be used to print text, the values of variables, as well as control the output screen appearance. Some of the most common DEBUG commands are shown in Table 3.

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG “Text to display”</td>
<td>Prints the text between quotes to the debug window</td>
</tr>
<tr>
<td>DEBUG DEC variable</td>
<td>Prints the value of variable to the debug window</td>
</tr>
<tr>
<td>DEBUG CR</td>
<td>“CR” is the carriage return character. It moves the cursor to the next line</td>
</tr>
<tr>
<td>DEBUG CLR</td>
<td>Clears the debug window</td>
</tr>
</tbody>
</table>

A number of the commands shown in Table 3 can be placed in one line of code. To place multiple commands on one line, separate them with a comma. An example of using the **DEBUG**
command is shown below with the resulting output is shown in Figure 5.

```
variable VAR byte
variable = 27
DEBUG CLS
DEBUG "The value of the variable is: ", DEC variable, CR
DEBUG "This is the second line."
DEBUG "Variable = ", DEC variable
```

### 2.2.6 Labels and GOTO

In PBASIC, labels are used as bookmarks in the program; they do not perform any function other than marking a specific point in the program that can be referenced by other commands. To create a label, simply add a colon (:) after the word label. The word used as the label name must not be a protected word in PBASIC; in other words, it should not be a PBASIC command. Labels are most commonly used with the GOTO command. An example of this is shown below.

```
Label:
    program code here
GOTO Label
```

**Figure 5:** Debug Window Resulting from Above Commands

In this example, the *program code* will run; then, once the program reaches the GOTO command, it jumps back to the Label: thereby re-running the *program code*. This loop would continue indefinitely, provided there are no commands to end the program in the code between Label: and GOTO. It is important to note that PBASIC runs linearly, and that labels have no effect on this; labeled code is not treated as a sub-function.
2.2.7 GOSUB... RETURN

Subroutines are used to organize and simplify the execution of a program. A subroutine should perform a complete task such as sort a list, run a motor in a certain sequence, or measure the distance to an object. Subroutines are often used several times in a program, at different locations in the code. This helps reduce the size of the program by not having to write the same exact code several times in the program.

Like GOTO, the GOSUB command is a method jump to other areas of the program, defined by labels in the code. The major difference between GOSUB and GOTO is the RETURN command, which must be used with GOSUB. Once sent to a label via the GOSUB command and issued the RETURN command, the program will return to the line of code following the initial GOSUB declaration. An example demonstrating the use of the GOSUB... RETURN command is shown below.

```plaintext
'jump to run_motor subroutine
GOSUB run_motor

'returns to this line from the subroutine
PAUSE 2000

run_motor:
    commands to run motor

RETURN 'now return to main program
```

2.2.8 FOR... NEXT

The FOR... NEXT command allows a loop to be executed a specified number of times. The syntax is below.

```plaintext
FOR counter = StartValue TO EndValue
    program code
NEXT
```

Using this syntax, `counter` is a variable, usually a byte or a word, used as counter. The `StartValue` is the starting value of the `counter` variable, and the `EndValue` is the ending value of the counter. The loop is repeated until the value of `counter` is outside the range defined by `StartValue` and `EndValue`. The `program code` will be executed each time through the loop. You can also adjust the step size of the `counter` variable using the `STEP` command. The `STEP` value can also be negative.

An example of using the FOR... NEXT command is shown below. The code will simply print the value of `counter` to the debug window each time through the two loops.

```plaintext
counter VAR byte

counter = 1 TO 10
FOR counter = 1 TO 10
    DEBUG "Counter = ", DEC counter, CR
NEXT
```
'count down from 10 to 1
FOR counter = 10 TO 1 STEP -1
    DEBUG "Counter = ", DEC counter, CR
NEXT

2.2.9 IF... THEN

The IF... THEN command is used to conditionally execute commands. The syntax for the IF... THEN command is shown below.

If condition THEN
    program code
ELSEIF condition THEN
    program code
ELSE
    program code
ENDIF

The available logical comparisons are shown in Table 4. In its simplest form, the IF... THEN command can be used to redirect code execution to a label. An example of this is shown below, where if the value of variable is less than 4000, the code will jump to label. Keep in mind that if the value of variable is greater than 4000, the next line of code will be run. In this example that would still be the code following Label:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not Equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Great Than or Equal To</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal To</td>
</tr>
</tbody>
</table>

variable VAR word

IF variable < 4000 THEN Label

Label:

code to run if variable < 4000

More complex conditionals can be formed by using the ELSEIF and ELSE commands. An example of a more complex IF... THEN command is shown below.

variable VAR word

IF (variable > 4000) THEN
    DEBUG "Variable > 4000", CR
ELSEIF (variable = 4000) THEN

TABLE 4: Logical Comparisons in PBASIC
DEBUG "Variable = 4000", CR
ELSE
DEBUG "Variable < 4000", CR
ENDIF

2.2.10 DO... LOOP

The DO... LOOP command allows a loop to be repeated indefinitely, until an EXIT command is issued, or until a condition is met. In order to run a loop indefinitely, the syntax below is used.

DO
  program code
LOOP

An EXIT command can be issued inside the DO... LOOP command to end the execution of the loop and move to the next line of code after the LOOP portion of the code. An example of this is below, where if variable is equal to ten, then the loop ends.

variable VAR byte
DO
  IF variable = 10 THEN EXIT
LOOP

The DO... LOOP command can also be used with the WHILE or UNTIL statements to execute the loop until a condition is met. The conditionals are the same as used in the IF... THEN command and shown in Table 4. An example of the WHILE and UNTIL commands is shown below.

variable_one VAR byte
variable_two VAR byte

DO WHILE variable_one < 3
  program code
LOOP

DO
  program code
LOOP UNTIL variable_two > 5

The main difference between the WHILE and UNTIL commands is when the conditional statement is evaluated. The conditional for the WHILE is evaluated before the loop is run, and the conditional for the UNTIL command is run after the program code within the loop has run. In the example above, the value of variable_one is checked, and if it is less than three, then the program code is run. In the second loop, the program code is run; then, if variable_two is greater than five, the loop is run again.
2.3.11 MAINIO AND AUXIO

The BASIC Stamp BS2p40 has two sets of input/output (I/O) pins. The MAINIO and AUXIO commands distinguish which set of pins the program commands will effect. At startup the main I/O pins are the ones affected by any program commands. To program the auxiliary pins, the AUXIO command must be issued. Any commands following the AUXIO command will affect the auxiliary pins. To return to programming the main I/O pins, the MAINIO command must be issued. An example of switching between controlling main and auxiliary pins is shown below.

- **HIGH** 11  ‘set main I/O pin 11 high
- **LOW** 9   ‘set main I/O pin 9 low
- **AUXIO**  ‘switch to auxiliary I/O
- **LOW** 11  ‘set auxiliary I/O pin 11 low
- **HIGH** 8   ‘set auxiliary I/O pin 8 high
- **MAINIO**  ‘switch back to main I/O
- **TOGGLE** 11 ‘toggle main I/O pin 11 low
- **HIGH** 9   ‘set main I/O pin 9 high
The controller box is also equipped with timer, which is controlled using the SEROUT command to an AVR microcontroller which is connected to the auxiliary I/O of the BASIC Stamp. Once the commanded time is up, the microcontroller drives pin 11 high. The syntax for the commands for time is shown below.

```
'use the auxiliary I/O pins
AUXIO

'begin timer
SEROUT 10, 247, [0, config, counter]

'continually check the value of pin 11
check:

'when pin 11 is high,
'jump to the time_up label
IF IN11 = 1 THEN time_up

GOTO check

time_up:

program code to run once timer is finished
```

In this code, the “0” part of the command resets the timer; it should not be changed. The config variable that defines how fast the timer should count. The possible count speeds and the corresponding values of the config variable are shown in Table 5. The counter variable defines the number of increments to count. When this number of counts is reached, pin 11 is driven high.

<table>
<thead>
<tr>
<th>config VALUE</th>
<th>INCREMENT SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1ms</td>
</tr>
<tr>
<td>2</td>
<td>10ms</td>
</tr>
<tr>
<td>4</td>
<td>100ms</td>
</tr>
<tr>
<td>8</td>
<td>250ms</td>
</tr>
</tbody>
</table>
This chapter will explain the electromechanical components of the kit, both sensors and actuators. Sensors include switches, an infrared distance sensor, and an encoder. Actuators include DC motors, a stepper motor, and solenoids. For each component, its basic operation will be discussed; then, how to program the controller box to utilize each component will be covered.
3.1.1 Switches

The switches provided are shown in Figure 6. They are simply two wires connected by a mechanically-triggered attachment. In the relaxed state, the two wires are said to be open, not connected. When the lever is pressed down, closing the switch, the two wires are shorted and a current flows. The long-arm switch is momentary, its circuit only remains closed while the lever is pressed. The pushbutton switch is a toggle switch. Once pressed, it will remain in the switched position until pressed again.

The switches provide an input signal in one of two possible states. A “1” corresponds to a closed circuit, the condition when the switch is closed, while a “0” corresponds to an open circuit, which is the condition when the switch is not pressed.
Each pin location has a corresponding register that holds the state of that port, either a 0 or a 1. In order for the BASIC Stamp to determine the state of a switch, the corresponding register should be read. Refer to Table 1 for the box connection to PBASIC translation. In order for a program to act upon the state of the switch, and IF... THEN command should be used. An example of this is below.

```
check:

  'If switch connected to IN1 is pressed, jump to Out_of_Loop
  IF IN4 = 1 THEN Out_of_Loop

GOTO check

Out_of_Loop:

  program code to run once switch is pressed
```

### 3.1.2 INFRARED DISTANCE SENSOR

The Infrared (IR) Distance Sensor, shown in Figure 7, is used to detect objects within a given range, from approximately 4 to 31 inches for the sensor in the kit. To determine the distance of an object, the sensor first sends an IR beam; it then receives this beam, once reflected from an object. The time between when the beam was sent and when it was received is used to calculate the distance of the object.

It is important that the wires from the IR sensor are connected to the correct ports on the controller box. The proper way to wire the IR sensor is shown in Figure 8.

The AVR microcontroller that controls the IR distance sensor is connected to the auxiliary I/O ports of the BASIC Stamp, so the AUXIO command must be issued to communicate with it. The syntax for communication with the IR sensor is below.

```
distance VAR Byte

AUXIO 'use the auxiliary I/O pins

'initialize IR sensor
SEROUT 14, 247, [1]

'retrieve IR sensor reading
```
In this command, the first SEROUT command initializes the AVR, and the SERIN command receives the distance reading. The distance variable returned will be a number between 0 and 255, indicating distance. The distance does not necessarily vary linearly with the value of the variable.

3.1.3 THE ENCODER

The encoder is shown in Figure 9 and is used to measure rotational motion. The rotational motion of the shaft of the encoder is converted to electrical pulses, or counts, that can be interpreted by a microcontroller. The rotational measurement is based upon the number of counts. The encoder included in the kit is a relative encoder, meaning it only counts relative to its starting position for each count command.

It is connected to the controller box via a three-prong connector and should be wired to the two outside ports of the connector. The encoder is connected to auxiliary I/O of the BASIC Stamp, so the AUXIO command must be issued to control it. The command for the encoder contains two main steps, beginning a count and checking the status of pin 13 to determine when the count is finished. The syntax is below.

```
enc_count VAR Byte
enc_count = 15

‘use the auxiliary I/O pins AUXIO
```
Main:
SEROUT 12, 247, [0,enc_count]

check:
'check status of Pin 13
IF IN13 = 1 THEN

    DEBUG "count finished", CR
    GOTO main

ELSE

    'if pin 13 is not high, check again
    GOTO check

ENDIF

In this code, the SEROUT command begins counting encoder pulses. The code after the label check: checks the status of pin 13, which is driven high when the number of encoder pulses set by the enc_count variable is met. The status of the pin is repeatedly checked, and once it is driven high, “count finished” is printed to the Debug window and a new count is begun by jumping back to the beginning of the program using the GOTO command.

Note that the direction of rotational motion has no effect on the number of pulses counted. A count in the clockwise direction is equal to a count in the counter-clockwise direction. So, in the code above, 15 counts in the clockwise direction will send pin 13 high, 15 counts in the counter-clockwise direction will send pin 13 high, or some combination of counts in either direction totaling 15 (e.g. 7 clockwise and 8 counter-clockwise) will send pin 13 high.
3.2.1 DC MOTORS

One of the two types of motors provided is a brushed DC motor; the two issued in the kit will be similar to those shown in Figure 10. The actual motors issued may differ, but their functionality and programming is the same.

The DC motor works by inducing an electromagnetic field using coils of wire around an armature. Around the inside of the casing are permanent magnets, and as the shaft turns, the brushes reverse the polarity of the armature, causing the shaft to rotate.

There are two ports on the controller box that can be used to control the DC motors, which are connected to the main I/O portion of the BASIC Stamp, on pins 14 and 15. The motor con-
nection to pin 14 is labeled as DCM1 on the box. The connection to pin 15 is labeled as DCM2. For each, a SEROUT command is sent to an AVR microcontroller that drives the motor. The speed, direction, and running time of the motor are specified through the SEROUT command to the microcontroller. The syntax is shown below.

' must be controlling main I/O...
' If not already, issue MAINIO command

'DC Motor 1 - DCM1
SEROUT 14, 247, [speed, direction, time]

'DC Motor 2 - DCM2
SEROUT 15, 247, [speed, direction, time]

In this command, direction sets the direction of the motor and should be set to 0 or 1; anything other than 0 will be interpreted as 1. The time is the number of 0.25 second intervals that the motor should run. Setting time to zero will cause the motor to run indefinitely. To stop a motor set to run indefinitely, reissue the SEROUT command with speed set to zero.

The speed portion of the command determines the speed at which the motor should run. The speed should be set between 10 and 240, with higher numbers indicate higher speeds. Also, notice that the first number after the SEROUT command in the commands above selects which motor to control. DC motor one, labeled DCM1 on the controller box, corresponds to this number being 14 and DC motor two, labeled DCM2, corresponds to it being 15.

3.2.2 Stepper Motors

The second type of motor provided is a stepper motor, shown in Figure 11. It should be wired according to the diagram in Figure 12. A stepper motor works differently than the DC motor and is preferable in certain situations. For example, the stepper motor requires (effectively) no time to ramp up or down from its current speed. Also, the speeds are easier to control and are more repeatable than the DC motor. A disadvantage of the stepper motor is that when turned off, the shaft is no longer held in place by the stator and is free to rotate. More information on the stepper motors included in the kit can be found at:
The stepper motor is connected to the main I/O of the BASIC Stamp through an AVR microcontroller at pin 13 and is controlled using a SEROUT command. The syntax is shown below.

```
'SEROUT        13, 247, [speed, direction, step_count, multiplier]
```

In this command, the `speed` should be set to a number between 0 and 100. A `speed` of 0 stops the motor and 100 is maximum speed. The motor speed varies linearly between these values dependent on the value of `speed`. The `direction` variable should be either 0 or 1, where anything other than 0 will be interpreted as 1.

The `step_count` and `multiplier` variables define the number of steps that the motor will turn. For example, if the step count is set to 200, or one full revolution, and the multiplier is set to 50, the motor will make 50 full revolutions.

### 3.2.3 Solenoids

A solenoid is essentially a coil of wire around a metallic plunger. When current is run through the coil of wire, it acts as an electromagnet, pulling in the plunger. The two solenoids issued as part of the kit are shown in Figure 13.

Solenoids should be connected to the digital outputs of the controller box, labeled DOUT1-DOUT4. They are controlled
using the HIGH, LOW, and TOGGLE commands discussed in Section 2.3.3. Issuing the HIGH command activates the solenoid, pulling in the plunger. LOW deactivates the solenoid, but will not return the plunger to its original position. Example code for the solenoids is shown below.

```
' must be controlling main I/O
' If not already, issue MAINIO command

'turn on solenoid connected to DOUT4
HIGH 11

'turn on solenoid connected to DOUT1
HIGH 8

' pause for 2 seconds
PAUSE 2000

'turn off solenoid connected to DOUT1
LOW 8

'toggle the state of DOUT4
' here it turns off
TOGGLE 11
```
The pneumatics system, shown in Figure 14, centers around pneumatic actuators, which can be repeatedly extended. This section will describe the hardware components of the kit and how to program the controller box to use the system.
The pneumatics system includes a tank, shown in Figure 15, and pneumatic actuators, shown in Figure 16, as well as several connecting fittings and tubing. Some combination of the two types of valves shown in Figure 17 are used to control the actuator. In order to use the pneumatic system, it must be connected properly. If it is not correctly connected, air will escape, preventing the system from being pressurized.

A bicycle pump can be used to provide the pressure needed for the system, up to a maximum of 100 psi. It will not operate properly if the pressure drops below 25 psi. Higher pressure allows more actuations of the cylinder to be achieved before a re-pressurization is needed. However, **DO NOT OVER PRESSURIZE THE SYSTEM.**

The pump connects to the tank using a 4mm diameter piece of tubing, through the 4mm one-way coupler on the back of the pressure gauge. After filling the tank, the 4mm tube should be removed from the tank prior to being removed from the air source. Leaving the tube in the tank keeps the one-way valve open, allowing the air to escape.

For the larger of the two valve types, the correct connection is shown in Figure 18. The pneumatic valve controls the airflow of the system; it is a solenoid valve with open and closed states. In the closed state, ports 1 and 2 are connected. Because port 2 is plugged, the system remains pressurized. Once the valve is opened, port 1 is connected to port 4, and air flows from the tank, extending the actuator. When the valve returns to the closed position, port 1 is again connected to the plugged port 2. Port 4 is also connected to port 3, which is open. The spring return in the actuator returns it to its un-extended position, pushing air out of the open port 3.
The correct connection of the smaller valve is shown in Figure 19. Its connection is much simpler, but its operation is essentially the same as the larger valve.
Figure 19: Connection of Pneumatic System with the Small Valve
Pneumatic Programming

The pneumatic valve is a solenoid and is programmed identically to the solenoids issued in the kit, using the **HIGH**, **LOW**, and **TOGGLE** commands. The valve can be connected to any one of digital output pins 8 to 11, labeled DOUT1 - DOUT4 on the controller box. Remember that these ports are on the main I/O and that commands must be addressing these ports. An example of controlling the pneumatic valve, connected to DOUT2 (main I/O pin 9), is below.

```
counter VAR Byte

' must be controlling main I/O...
' If not already, issue MAINIO command

' open valve and extend actuator
HIGH 9  ' MAINIO Pin 9 = DOUT2

PAUSE 1000 ' pause 1 second

' close valve and retract actuator
LOW 9

'Toggle the state of the valve 5 times
FOR counter = 1 TO 5
   TOGGLE 9

' Pause 1 second between toggles
PAUSE 1000

NEXT
```

In this code example, the valve is opened using the **HIGH** command, extending the cylinder. After a one second pause, the **LOW** command is issued, returning the valve to the closed position. When this happens, the spring return retracts the actuator. Next, the code enters a **FOR... NEXT** loop which toggles the state of the valve five times, with a one second pause between each toggle.