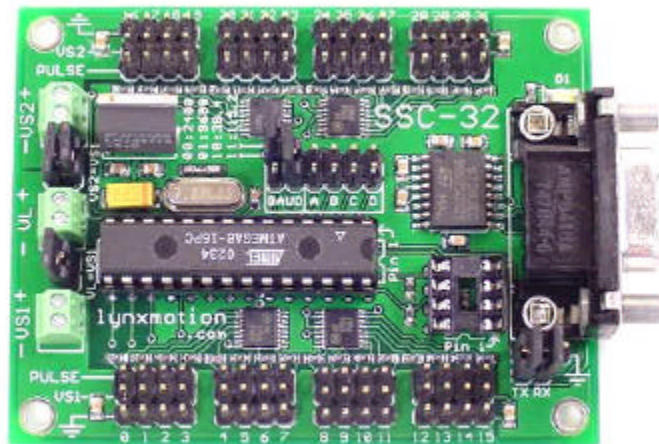




# SSC-32 Ver 2.0

Manual written for firmware  
version SSC32-1.03XE  
Range is 0.50mS to 2.50mS



## **Lynxmotion, Inc.**

PO Box 818  
Pekin, IL 61555-0818  
Tel: 309-382-1816 (Sales)  
Tel: 309-382-2760 (Support)  
Fax: 309-382-1254  
E-m: [sales@lynxmotion.com](mailto:sales@lynxmotion.com)  
E-m: [tech@lynxmotion.com](mailto:tech@lynxmotion.com)  
Web: <http://www.lynxmotion.com>

Users Manual SSC-32 Ver 2.0

## Things that go Boom!



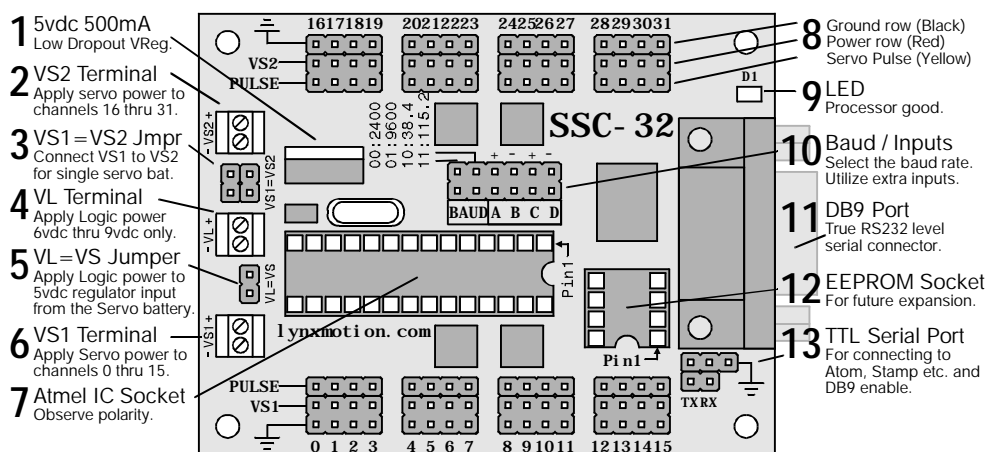
**B** Caution! Read this quick start guide completely before wiring and applying power to the board! Errors in wiring can damage the SSC-32 board, Atmel or EEPROM Chip, and any attached servos or peripherals.



**B** Caution! Never reverse the power coming in to the board. Make sure the black wire goes to (-) ground, and the red wire goes to (+) Vlogic, or Vservo. Never connect peripherals when the board is powered on.



**B** Caution! The onboard regulator can provide 250mA total. This includes the microcontroller chip, the onboard LEDs, and any attached peripherals. Drawing too much current can cause the regulator to overheat.



**1** The Low Dropout regulator will provide 5vdc out with as little as 5.5vdc coming in. This is important when operating your robot from a battery. It can accept a maximum of 9vdc in. The regulator is rated for 500mA, but we are de-rating it to 250mA to prevent the regulator from getting too hot.

**2** This terminal connects power to servo channels 16 thru 31. Apply 4.8vdc to 7.2vdc for normal servos. Apply 4.8vdc to 6.0vdc when using micro servos. Do not exceed 7.4vdc (measure it) when using HSR-5995TG servos!

Board	Input
VS2 +	RED
VS2 -	BLACK

**3** These jumpers are used to connect VS1 to VS2. Use this option when you are powering all servos from the same battery. Use both jumpers.

**4** This is the Electronics Power Input. It is also referred to as the Logic Voltage, or VL. This input is normally used with a 9vdc battery connector to provide power to the ICs and anything connected to the 5vdc lines on the board. This input is used to isolate the logic from the Servo Power Input.

Board	Input
VL +	RED
VL -	BLACK

**5** This jumper allows powering the microcontroller and support circuitry from the servo power supply. This requires at least 6vdc to operate correctly. If the microcontroller resets when many servos are moving it may be necessary to power the microcontroller separately using the VL input. A 9vdc battery works nicely for this.

**6** This terminal connects power to servo channels 0 thru 15. Apply 4.8vdc to 7.2vdc for normal servos. Apply 4.8vdc to 6.0vdc when using micro servos. Do not exceed 7.4vdc (measure it) when using HSR-5995TG servos!

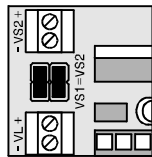
Board	Input
VL1 +	RED
VL1 -	BLACK

**7** This is where the Atmel IC chip goes. Be careful to insert it with Pin 1 in the upper right corner as pictured. Take care not to bend the pins.

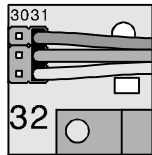
- 8** This is where you connect the servos or other output devices. Use caution and remove power when connecting anything to the I/O bus.
- 9** This is the Processor Good LED. It will light steady when power is applied and will remain lit until the processor has received a valid serial command. It will then go out and then blink whenever it is receiving serial data. Note, this feature may not be used on user-submitted firmware for the SSC-32.
- 10** The two BAUD inputs allow configuring the baud rate. Please see the examples below. The ABCD inputs have both static and latching support. The inputs have internal weak (50k) pullups that are used when a Read Digital Input command is used. A normally open switch connected from the input to ground will work fine. These features may not be used on user-submitted firmware for the SSC-32.
- | Jumpers | Baud Rate |
|---------|-----------|
| 0 0     | 2400      |
| 0 1     | 9600      |
| 1 0     | 38.4k     |
| 1 1     | 115.2k    |
- 11** Simply plug a straight-through M/F DB9 cable from this plug to a free 9 pin serial port on your PC for receiving servo positioning data. Alternately a USB-to-serial adaptor will work well.
- 12** This is an 8 pin EEPROM socket. It is not used in this version of the firmware, although it will be used in future versions.
- 13** This is the TTL serial port or DB9 serial port enable. Install two jumpers as illustrated below to enable the DB9 port. Install wire connectors to utilize TTL serial communication from a host microcontroller.

### Shorting Bar Jumpers and Connectors at a glance

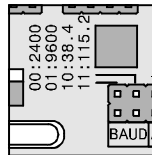
Applies VS1 to VS2.



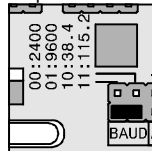
Example servo connection 16-31.



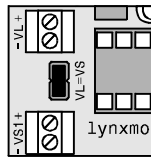
Baud rate 2400.



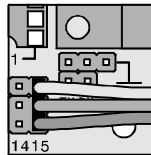
Update the Atmel chip firmware.



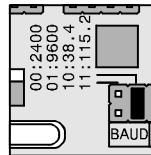
Applies VS to VL.



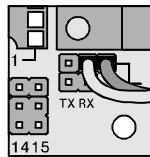
Example servo connection 0-15.



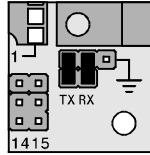
Baud rate 9600.



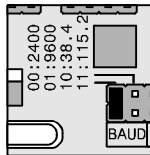
TTL Serial com. SSC-32 side...



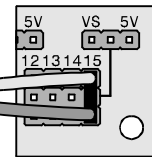
DB9 enable for PC use.



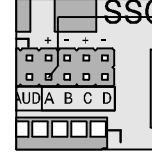
Baud rate 38.4k for Basic Atom use.



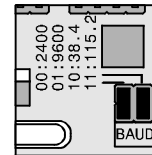
TTL Serial com. Bot Board side...



ABCD auxiliary inputs.



Baud rate 115.2k for PC use.



Caution! Don't do this if you don't know what you're doing. Connecting this jumper can overwrite the Atmel chip's firmware.

## Getting Started

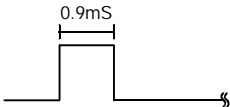

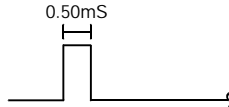

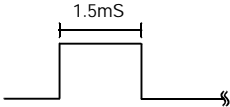

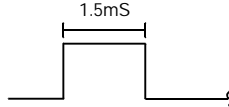

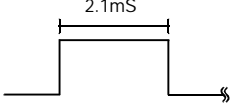

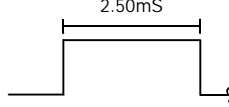

### What is a Servo?

Before we illustrate how to use the servo controller we need to explain what a servo is, and define the control methodology.

Pulse-proportional servos are designed for use in radio-controlled (R/C) cars, boats and planes. They provide precise control for steering, throttle, rudder, etc. using a signal that is easy to transmit and receive. The signal consists of positive going pulses ranging from 0.9 to 2.1mS (milliseconds) long, repeated 50 times a second (every 20mS). The servo positions its output shaft in proportion to the width of the pulse, as shown below.

In radio-control applications, a servo needs no more than a 90° range of motion, since it is usually driving a crank mechanism that can't move more than 90°. So when you send pulses within the manufacturer-specified range of 0.9 to 2.1mS, you get around 90° range of motion.

Most servos have more than 90° of mechanical range. In fact, most servos can move up to 180° of rotation. However, some servos can be damaged when commanded past their mechanical limitations. The SSC-32 lets you use this extra range. A position value of 500 corresponds to 0.50mS pulse, and a position value of 2500 corresponds to a 2.50mS pulse. A one unit change in position value produces a 1uS (microsecond) change in pulse width. The positioning resolution is 0.09°/unit (180°/2000). From here on, the term pulse width and position are the same.

Normal Range	Extended Range
 0.9mS  Servo -45 °	 0.50mS  Servo -90 °
 1.5mS  Servo Centered	 1.5mS  Servo Centered
 2.1mS  Servo +45 °	 2.50mS  Servo +90 °

Remember that some servos may not be able to move the entire 180° range. Use care when testing servos. Move to the extreme left or right slowly, looking for a point when additional positioning values no longer result in additional servo output shaft movement. When this value is found, put it as a limit in your program to prevent damaging the servo. Generally, micro servos are not able to move the entire 180° range.

### What is Open Source?

It simply means we are distributing the source code for the bootloader and firmware. The goal is to have an affordable platform that many people will provide firmware for. It should also help many aspiring programmers learn some tricks. Anyone can use the source code to write specialized firmware, providing you allow Lynxmotion, Inc. to publish it for others to enjoy. The source code can not be used in a commercial product. As it is, this servo controller will outperform controllers costing two to three times as much. Having several "flavors" of the firmware will make this an even better value.

## Command Formatting for the SSC-32

### Command Types and Groups

- |                           |                                     |
|---------------------------|-------------------------------------|
| 1) Servo Movement.        | 7) Read Analog Inputs.              |
| 2) Discrete Output.       | 8) 12 Servo Hexapod Gait Sequencer. |
| 3) Byte Output.           | 9) Query Hex Sequencer.             |
| 4) Query Movement Status. | 10) Get Version.                    |
| 5) Query Pulse Width.     | 11) Go To Boot.                     |
| 6) Read Digital Inputs.   | 12) MiniSSC-II Compatibility.       |

With the exception of MiniSSC-II mode, all SSC-32 commands must end with a carriage return character (ASCII 13). Multiple commands of the same type can be issued simultaneously in a *Command Group*. All of the commands in a command group will be executed after the final carriage return is received. Commands of different types cannot be mixed in the same command group. In addition, numeric arguments to all SSC-32 commands must be ASCII strings of decimal numbers, e.g. "1234". Some commands accept negative numbers, e.g. "-5678". Programming examples will be provided. ASCII format is not case sensitive. Use as many bytes as required. Spaces, tabs, and line feeds are ignored.

### Servo Move or Group Move:

# <ch> P <pw> S <spd> ... # <ch> P <pw> S <spd> T <time> <cr>

- <ch> = Channel number in decimal, 0 - 31.
- <pw> = Pulse width in microseconds, 500 - 2500.
- <spd> = Movement speed in uS per second for one channel. (Optional)
- <time> = Time in mS for the entire move, affects all channels, 65535 max. (Optional)
- <cr> = Carriage return character, ASCII 13. (Required to initiate action)
- <esc> = Cancel the current command, ASCII 27.

Servo Move Example: "#5 P1600 S750 <cr>"

The example will move the servo on channel 5 to position 1600. It will move from its current position at a rate of 750uS per second until it reaches its commanded destination. For a better understanding of the speed argument consider that 1000uS of travel will result in around 90° of rotation. A speed value of 100uS per second means the servo will take 10 seconds to move 90°. Alternately a speed value of 2000uS per second equates to 500mS (half a second) to move 90°.

Servo Move Example: "#5 P1600 T1000 <cr>"

The example will move servo 5 to position 1600. It will take 1 second to complete the move regardless of how far the servo has to travel to reach the destination.

Servo Group Move Example: "#5 P1600 #10 P750 T2500 <cr>"

The example will move servo 5 to position 1600 and servo 10 to position 750. It will take 2.5 seconds to complete the move, even if one servo has farther to travel than another. The servos will both start and stop moving at the same time. This is a very powerful command. By commanding all of the legs in a walking robot with the Group Move it is easy to synchronize complex gaits. The same synchronized motion can benefit the control of a robotic arm as well.

You can combine the speed and time commands if desired. The speed for each servo will be calculated according to the following rules:

1. All channels will start and end the move simultaneously.
2. If a speed is specified for a servo, it will not move any faster than the speed specified (but it might move slower if the time command requires).
3. If a time is specified for the move, then the move will take at least the amount of time specified (but might take longer if the speed command requires).

Servo Move Example: "#5 P1600 #17 P750 S500 #2 P2250 T2000 <cr>"

The example provides 1600uS on ch5, 750uS on ch17, and 2250uS on ch2. The entire move will take at least 2 seconds, but ch17 will not move faster than 500uS per second. The actual time for the move will depend on the initial pulse width for ch17. Suppose ch17 starts at position 2000. Then it has to move 1250uS. Since it is limited to 500uS per second, it will require at least 2.5 seconds, so the entire move will take 2.5 seconds. On the other hand, if ch17 starts at position 1000, it only needs to move 250uS, which it can do in 0.5 seconds, so the entire move will take 2 seconds.

Important! Don't issue a speed or time command to the servo controller as the first instruction. It will assume it needs to start at 500uS and will zip there as quickly as possible. The first positioning command should be a normal "# <ch> P <pw>" command. Because the controller doesn't know where the servo is positioned on power up it has to be this way.

#### **Pulse Offset:**

# <ch> PO <offset value> ... # <ch> PO <offset value> <cr>

<ch> = Channel number in decimal, 0 - 31.

<offset value> = 100 to -100 in uSeconds.

<cr> = Carriage return character, ASCII 13.

The servo channel will be offset by the amount indicated in offset value. This makes it easy to setup legs in a robot that do not allow mechanical calibration.

#### **Discrete Output:**

# <ch> <lvl> ... # <ch> <lvl> <cr>

<ch> = Channel number in decimal, 0 - 31.

<lvl> = Logic level for the channel, either 'H' for High or 'L' for Low.

<cr> = Carriage return character, ASCII 13.

The channel will go to the level indicated within 20mS of receiving the carriage return.

Discrete Output Example: "#3H #4L <cr>"

This example will output a High (+5v) on channel 3 and a Low (0v) on channel 4.

**Byte Output:**

# <bank> : <value> <cr>

<bank> = (0 = Pins 0-7, 1 = Pins 8-15, 2 = Pins 16-23, 3 = Pins 24-31.)

<value> = Decimal value to output to the selected bank (0-255). Bit 0 = LSB of bank.

This command allows 8 bits of binary data to be written at once. All pins of the bank are updated simultaneously. The banks will be updated within 20mS of receiving the CR.

Bank Output Example: "#3:123 <cr>"

This example will output the value 123 (decimal) to bank 3. 123 (dec) = 01111011 (bin), and bank 3 is pins 24-31. So this command will output a "0" to pins 26 and 31, and will output a "1" to all other pins.

**Query Movement Status:**

Q <cr>

This will return a "." if the previous move is complete, or a "+" if it is still in progress.

There will be a delay of 50uS to 5mS before the response is sent.

**Query Pulse Width:**

QP <arg> <cr>

This will return a single byte (in binary format) indicating the pulse width of the selected servo with a resolution of 10uS. For example, if the pulse width is 1500uS, the returned byte would be 150 (binary).

Multiple servos may be queried in the same command. The return value will be one byte per servo. There will be a delay of least 50uS to 5mS before the response is sent. Typically the response will be started within 100uS.

**Read Digital Inputs:**

A B C D AL BL CL DL <cr>

A, B, C, or D reads the value on the input as a binary value. It returns ASCII "0" if the input is a low (0v) or an ASCII "1" if the input is a high (+5v).

AL, BL, CL, or DL returns the value on the input as an ASCII "0" if the input is a low (0v) or if it has been low since the last \*L command. It returns a high (+5v) if the input is a high and never went low since the last \*L command. Simply stated it will return a low if the input ever goes low. Reading the status automatically resets the latch.

The ABCD inputs have a weak pullup (~50k) that is enabled when used as inputs. They are checked approximately every 1mS, and are debounced for approximately 15mS. The logic value for the read commands will not be changed until the input has been at the new logic level continuously for 15mS. The Read Digital Input Commands can be grouped in a single read, up to 8 values per read. They will return a string with one character per input with no spaces.

Read Digital Input Example: "A B C DL <cr>"

This example returns 4 characters with the values of A, B, C, and D-Latch. If A=0, B=1, C=1, and DL=0, the return value will be "0110".

#### **Read Analog Inputs:**

VA VB VC VD <cr>

VA, VB, VC, VD reads the value on the input as analog. It returns a single byte with the 8-bit (binary) value for the voltage on the pin.

When the ABCD inputs are used as analog inputs the internal pullup is disabled. The inputs are digitally filtered to reduce the effect of noise. The filtered values will settle to their final values within 8mS of a change. A return value of 0 represents 0vdc. A return value of 255 represents +4.98vdc. To convert the return value to a voltage multiply by 5/256. At power-up the ABCD inputs are configured for digital input with pullup. The first time a V\* command is used, the pin will be converted to analog without pullup. The result of this first read will not return valid data.

Read Analog Input Example: "VA VB <cr>"

This example will return 2 bytes with the analog values of A and B. For example if the voltage on Pin A is 2vdc and Pin B is 3.5vdc, the return value will be the bytes 102 (binary) and 179 (binary).

#### **12 Servo Hexapod Sequencer Commands:**

LH <arg>, LM <arg>, LL <arg>

Set the value for the vertical servos on the left side of the hexapod. LH sets the high value, i.e. the pulse width to raise the leg to its maximum height; LM sets the mid value; and LL sets the low value. The valid range for the arguments is 500 to 2500uS.

RH <arg>, RM <arg>, RL <arg>

Set the value for the vertical servos on the right side of the hexapod. RH sets the high value, i.e. the pulse width to raise the leg to its maximum height; RM sets the mid value; and RL sets the low value. The valid range for the arguments is 500 to 2500uS.

VS <arg>

Sets the speed for movement of vertical servos. All vertical servo moves use this speed. Valid range is 0 to 65535uS/Sec.

LF <arg>, LR <arg>

Set the value for the horizontal servos on the left side of the robot. LF sets the front value, i.e. the pulse width to move the leg to the maximum forward position; LR sets the rear value. The valid range for the arguments is 500 to 2500uS.

RF <arg>, RR <arg>

Set the value for the horizontal servos on the right side of the robot. RF sets the front value, i.e. the pulse width to move the leg to the maximum forward position; RR sets the rear value. The valid range for the arguments is 500 to 2500uS.

HT <arg>

Sets the time to move between horizontal front and rear positions. The valid range for the argument is 1 to 65535uS.



XL <arg>, XR <arg>

Set the travel percentage for left and right legs. The valid range is -100% to 100%. Negative values cause the legs on the side to move in reverse. With a value of 100%, the legs will move between the front and rear positions. Lower values cause the travel to be proportionally less, but always centered. The speed for horizontal moves is adjusted based on the XL and XR commands, so the move time remains the same.

XS <arg>

Set the horizontal speed percentage for all legs. The valid range is 0% to 200%. With a value of 100%, the horizontal travel time will be the value programmed using the HT command. Higher values proportionally reduce the travel time; lower values increase it. A value of 0 will stop the robot in place. The hex sequencer will not be started until the XS command is received.

XSTOP

Stop the hex sequencer. Return all servos to normal operation.

Notes on Hex Sequencer:

1) The following servo channels are used for the Hex Sequencer

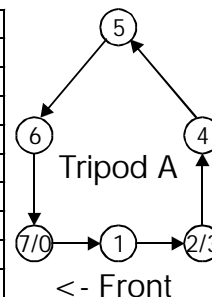
0 = Right Rear Vertical	16 = Left Rear Vertical
1 = Right Rear Horizontal	17 = Left Rear Horizontal
2 = Right Center Vertical	18 = Left Center Vertical
3 = Right Center Horizontal	19 = Left Center Horizontal
4 = Right Front Vertical	20 = Left Front Vertical
5 = Right Front Horizontal	21 = Left Front Horizontal

2) The Hexapod walking gait is an alternating tripod. The tripods are labeled Tripod A and Tripod B. Tripod A consists of {Left Front, Left Rear, Right Center}, and Tripod B consists of {Left Center, Right Front, Right Rear}.

3) While walking, the legs pass through 6 points: (Low Front), (Low Center), (Low Rear), (Mid Rear), (High Center), and (Mid Front). "Center" refers to the mid-point between the Front and Rear pulse widths.

4) The walking sequence consists of 8 states, numbered 0 – 7. They are defined below:

State	Tripod A		Tripod B	
	Vertical	Horizontal	Vertical	Horizontal
0	Low	Front to Center	Mid to High	Rear to Center
1	Low	Center to Rear	High to Mid	Center to Front
2	Low	Rear	Mid to Low	Front
3	Low to Mid	Rear	Low	Front
4	Mid to High	Rear to Center	Low	Front to Center
5	High to Mid	Center to Front	Low	Center to Rear
6	Mid to Low	Front	Low	Rear
7	Low	Front	Low to Mid	Rear



In this table, "Front" and "Rear" are modified by the XL and XR commands. A value of 100% results in the movement in the table. Between 0 and 100%, the Front/Rear positions are moved closer to Center. For negative values, Front and Rear are exchanged. For example, with an XL of -100%, in State 0, Tripod A on the left side would be moving Rear to Center, and Tripod B would be moving Front to Center.

- 5) When a horizontal servo is moving, its speed will be adjusted based on the Front/Rear pulse widths, the XL/XR percentage, and the XS percentage. Regardless of the travel distance from front to rear (adjusted by XL/XR), the total move time will be the HT divided by the XS percentage.
- 6) When a vertical servo is moving from Low to Mid or from Mid to Low, it will move at the speed specified by the VS command. When a vertical servo is moving from Mid to High or High to Mid, the vertical speed will be adjusted so that the horizontal and vertical movements end at the same time.
- 7) Any of the Hex Sequencer commands can be issued while the sequencer is operating. They will take effect immediately.

Hex Sequencer Examples:

"LH1000 LM1400 LL1800 RH2000 RM1600 RL1200 VS3000 <cr>"  
Sets the vertical servo parameters.

"LF1700 LR1300 RF1300 RR1700 HT1500 <cr>"  
Sets the horizontal servo parameters.

"XL50 XR100 XS100 <cr>"  
Causes the gradual left turn at 100% speed (and starts the sequencer if it is not already started).

"XL -100 XR 100 XS 50 <cr>"  
Causes a left rotate in place at 50% speed.

"XSTOP <cr>"  
Stops the sequencer and allows servo channels 0-5, 16-21 to be controlled using the normal servo movement commands.

#### **Query Hex Sequencer State:**

XQ <vr>  
Returns 1 digit representing the state of the hex sequencer, and the approximate percentage of movement in the state. The high nibble will be '0' to '7', and the low nibble will be '0' to '9'. For example, if the sequencer is 80% of the way through state 5, it will return the value 58 hex.

#### **Get Software Version:**

VER <cr>  
Returns the software version number as an ASCII string.

#### **Transfer to Boot:**

GOBOOT <cr>  
Starts the bootloader running for software updates. To exit the bootloader and start running the application, power cycle the control or enter (case sensitive, no spaces).

g0000<cr>

#### **SSC Emulation:** Binary format, 3-bytes.

Byte 1: 255, the sync byte

Byte 2: 0 - 31, the servo number

Byte 3: 0 - 250, the pulse width, 0=500uS, 125=1500uS, 250=2500uS

## Testing the Controller

The easiest way to test the controller is to use the Lynx SSC-32 Terminal. It's a free download from the website. Once installed click on the Port Drop Down and select your com port. This will work with USB to serial port adaptors. Install the jumpers for 115.2k baud and the two DB9 serial port enable jumpers. Plug a straight through DB9 M/F cable from the PC to the controller.

Install two servos, one on channel 0 and one on channel 1.

Power up the SSC-32 (Logic and Servo) and notice the green LED is illuminated.

Then click on the terminal window so you can type the following into it.

```
#0 P1500 #1 P1500 <cr>    <- (This means hit Enter.)
```

You should notice both servos are holding position in the center of their range. The LED is also no longer illuminated. It will now only light when the controller is receiving data. Type the following:

```
#0 P750 #1 P1000 T3000 <cr>
```

You should notice servo 0 moving CW slowly and servo 1 moving CCW a bit faster. They will arrive at their destination at the exact same time even though they are moving different distances.

Now to test the Query Movement Status. Type the following:

```
#0 P750 <cr>
```

Then type the following line. This will make the servo move full range in 10 seconds.

```
#0 P2250 T10000 <cr>
```

While the servo is moving type the following:

```
Q <cr>
```

When the servo is in motion the controller will return a "+". It will return a "." when it has reached its destination.

To experiment with the speed argument try the following:

```
#0 P750 S1000 <cr>
```

This will move the servo from 2250 to 750 (around 170°) in 1.5 seconds.

$$\frac{2250\mu\text{S} - 750\mu\text{S} \text{ (travel distance)}}{1000\mu\text{S/Sec. (speed value)}} = 1.5 \text{ Sec.}$$

Next try typing the following:

```
#0 P2250 S750 <cr>
```

This will move the servo from 750 to 2250 (around 170°) in 2.0 seconds.

$$\frac{2250\mu\text{S} - 750\mu\text{S} \text{ (travel distance)}}{750\mu\text{S/Sec. (speed value)}} = 2.0 \text{ Sec.}$$

Speed values above around 3500 will move the servo as quickly as the servo can move.

#### **Updating the SSC-32 firmware**

From the SSC-32 Terminal main screen click on Firmware. This will show the current version of the firmware at the top, and allow you to browse and open the new \*.abl firmware file. Click Begin Update to finish the update process.

Don't forget to check the website for the latest tutorials for the servo controller.

## Basic Atom Programming Examples for the SSC-32

```
' Atom / SSC-32 Test
' Configure the SSC-32 for 38.4k baud.

servo0pw var word
movetime var word

servo0pw = 1000
movetime = 2500

start:
servo0pw = 1000
serout p0,i38400,["#0P",DEC servo0pw,"T",DEC movetime,13]
pause 2500
servo0pw = 2000
serout p0,i38400,["#0P",DEC servo0pw,"T",DEC movetime,13]
pause 2500
goto start

' Biped example program.
aa var byte '<- general purpose variable.
rax var word '<- right ankle side-to-side. On pin0
ray var word '<- right ankle front-to-back. On pin1
rkn var word '<- right knee. On pin2
rhx var word '<- right hip front-to-back. On pin3
rhy var word '<- right hip side-to-side. On pin4
lax var word '<- left ankle side-to-side. On pin5
lay var word '<- left ankle front-to-back. On pin6
lkn var word '<- left knee. On pin7
lhx var word '<- left hip front-to-back. On pin8
lhy var word '<- left hip side-to-side. On pin9
ttm var word '<- time to take for the current move.

' First command to turn the servos on.
for aa=0 to 9
serout p0,i38400,["#", DEC2 aa\1, "P", DEC 1500, 13]
next

start:
' First position for step sequence, and time to move, put in your values here.
rax=1400: ray=1400: rkn=1400: rhx=1400: rhy=1400
lax=1400: lay=1400: lkn=1400: lhx=1400: lhy=1400
ttm=1000
gosub send_data
pause ttm

' Second position for step sequence, and time to move, put in your values here.
rax=1600: ray=1600: rkn=1600: rhx=1600: rhy=1600
lax=1600: lay=1600: lkn=1600: lhx=1600: lhy=1600
ttm=1000
gosub send_data
pause ttm

' Third...

' Forth...

' Etc...

goto start

' This sends the data to the SSC-32. The serout is all one line, no Wrap!
send_data:
serout p0,i38400,["#0P",DEC rax,"#1P",DEC ray,"#2P",DEC rkn,"#3P",DEC rhx,"#4P",DEC
rhy,"#5P",DEC lax,"#6P",DEC lay,"#7P",DEC lkn,"#8P",DEC lhx,"#9P",DEC lhy,"T",DEC
ttm,13]
return
```

Note: RESET is pulled up internal to ATMEGA8P

