Lynxmotion

Imagine it. Build it. Control it.™

Lynxmotion SSC-32U USB Servo Controller Board

Revisions

V1.2 July 2017

Feedback / Errata / Support:
http://www.robotshop.com/forum/ssc-32u-f142

Table of Contents

Overview

The Lynxmotion SSC-32U is a versatile and easy to use R/C servo controller, the core of which is an Atmel ATmega328p. Features include:

- USB, serial or XBee input
- Control up to 32 servo motors
- Built-in 12DoF hexapod sequencer
- 8 analog input pins
- Easy to understand command protocol
- Advanced features
- Ideal for use with robot arms or legged robots
- Compatible with FlowBotics Studio graphical software

Note that the SSC-32U is a dedicated servo controller; as such, the board is not meant to be "programmed" (or store code which a user creates) but instead simply receives and executes commands sent to it from an external device such as a computer or microcontroller. Using a dedicated servo controller frees up memory on a microcontroller which would be used to constantly update the servos' positions.

The SSC-32U is based largely on the SSC-32. The user guide for the SSC-32 can be found here for reference: http://www.lynxmotion.com/images/html/build136.htm

Digital vs Analog vs "Smart" Servos

Analog Servos

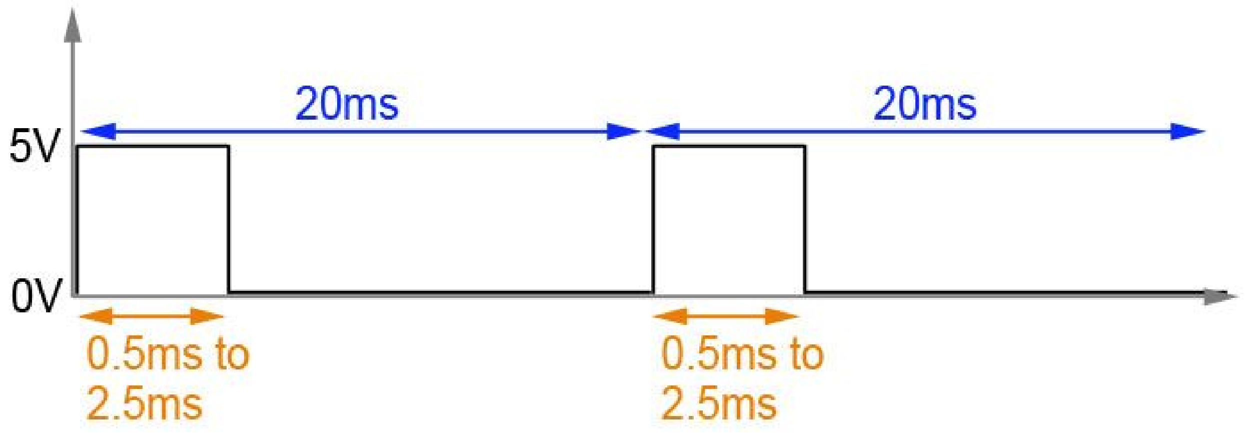
R/C Servo Basics

When dealing with a Remotely Controlled ("R/C") servo, it is important to know the following terminology:

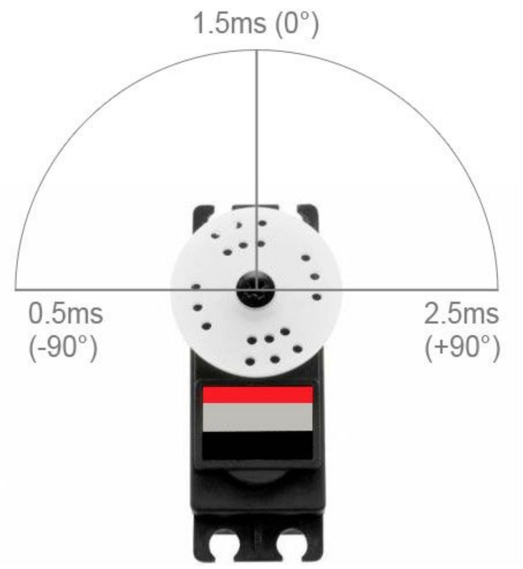
- Case (normally plastic, sometimes aluminum)
- DC motor (normally brushed, sometimes brushless or more exotic)
- Gears (which reduce the motor's speed and increase the torque)
- Spline (connection from the last internal gear; fits into the servo horn to make it rotate)
- Horn (the plastic or metal part which can be used to connect other items)
- Potentiometer (an absolute angel sensor used to provide the servo's position)
- Electronics (controls the motion)
- 3-pin cable and connector (0.1" standard spaced)

White / yellow / orange = Signal (Pulse)
Red = 4.8V to 6V (VS)
Black / brown = GND

R/C servos use a form of "Pulse Width Modulation" (PWM). In this case, the servo's electronics are waiting to receive a repeated 5V signal which it times to see how long that signal stays on. For most RC servos, a 500us long pulse (0.5ms) at 5V would rotate the servo horn to -90 degrees; a 2500us (2.5ms) long 5V pulse would rotate the horn to +90 degrees; therefore a 1500us pulse corresponds to 0 degrees (centered). This pulse needs to be repeated every ~20ms.



R/C Timed Pulse (repeated every 20ms)



Hitec Standard Servo Showing Operating Angle vs Signal

When an RC servo reaches a specific angle, it will try its best to retain that position, provided it keeps receiving the signal every 20ms. If the torque is higher than what the servo can provide, the current will increase drastically, causing the servo to heat up and eventually fail.

Most R/C servos have a duty cycle of around 25%. The “duty cycle” corresponds to how long the motor should be on versus off. This does not mean to say that you could run a servo motor for 1 hour and let it cool off for 3 hours; we suggest that a normal servo motor should be in constant operation for less than around 10 minutes, at which point it will ideally need around 30 minutes to cool off.

Continuous Rotation Servo

A continuous rotation RC servo uses the same pulses / control as a normal analog RC servo but instead converts the pulses to speed and direction, where 1.5ms corresponds to stopped, 0.5ms is full speed counter-clockwise and 2.5ms is full speed clockwise. Intermediate values corresponds to intermediate speeds. A continuous rotation servo cannot be assigned to a specific (or relative) angle and operates much like a DC gear motor.

Digital R/C Servo

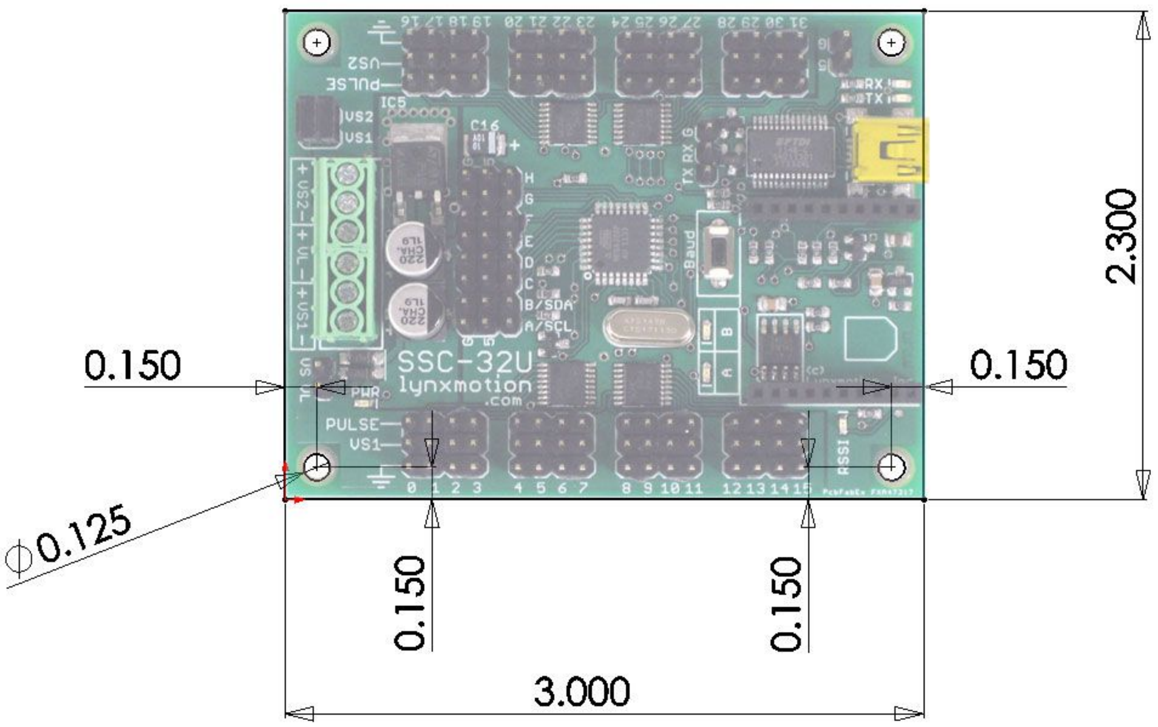
A digital servo motor operates the same way as a normal analog R/C servo, but often allows for more precise / faster movement. In many cases, the servo's limits and center position (as well as a variety of other paramaters) can be changed using a servo programmer (the SSC-32U is not a digital servo programmer). Once the servo has been programmed, it can be used just like any other analog R/C servo motor. Digital servo motors look the same as analog servo motors.

Smart Servo

A "Smart Servo" is a servo which is not normally controlled by an R/C signal but rather using serial commands. The connector of a smart servo is normally different than that of an R/C servo. The SSC-32U is not meant to control smart servo motors. Smart servo motors often have different / custom mechanical designs than R/C servos.

Hardware Information

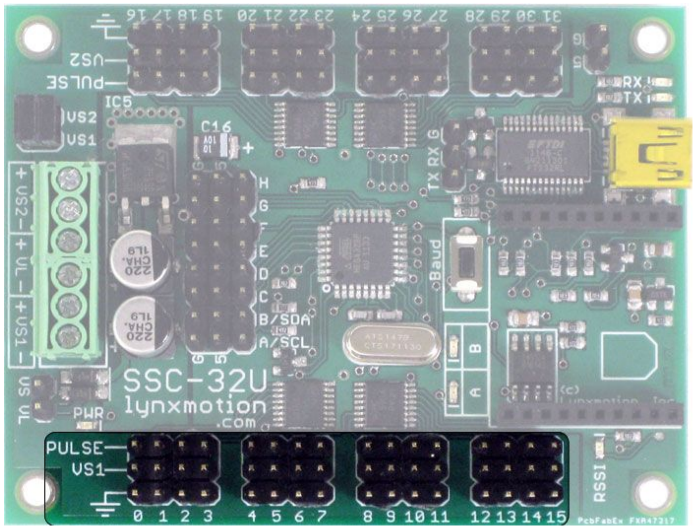
Dimensions



SSC-32U Board Dimensions (inches)

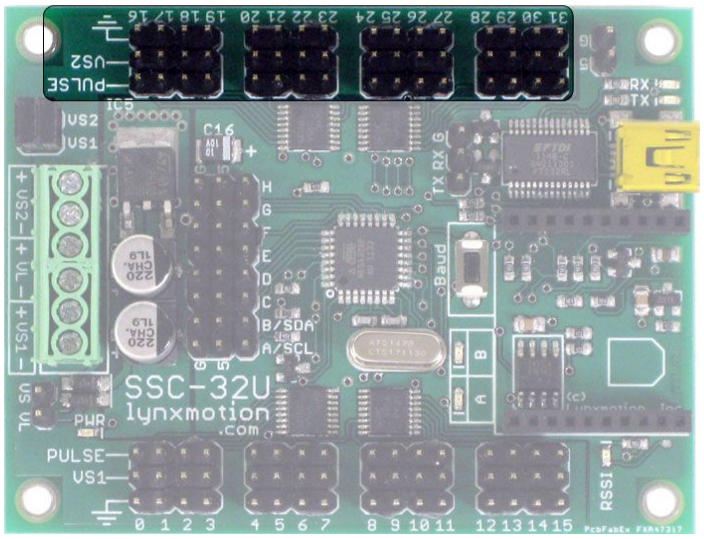
The SSC-32U is 3.00" x 2.30" with 0.125" holes set in 0.15" from each edge. It was designed to the same dimensions as the Lynxmotion BotBoarduino, the Lynxmotion SSC-32 (its predecessor), the Lynxmotion Bot Board and Bot Board 2 so they could be stacked.

S1 Pins



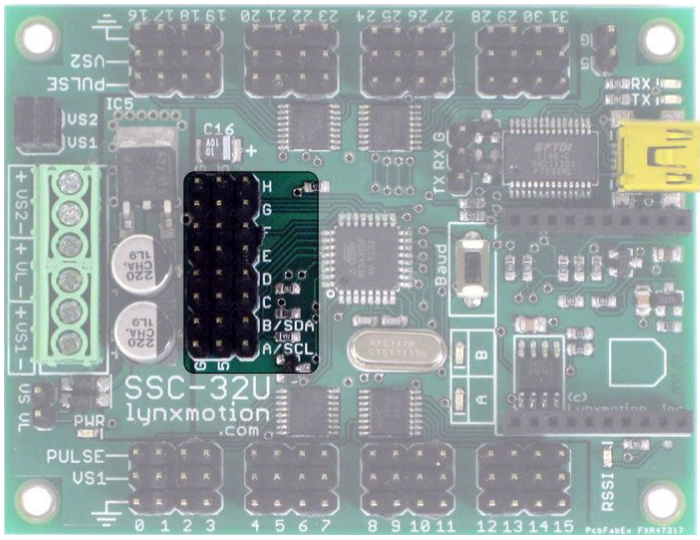
The SSC-32U servo controller can control up to 32 servos, which are laid out as two separate sections of 16 servos, each grouped together in sets of four. Pins 0 to 15 corresponds to VS1. The outermost pin (farthest from the center of the board) is the ground pin (normally the black wire in a three pin cable). The center pin corresponds to the voltage (normally the red wire) and is marked with VS1. The last pin is the signal pin which sends the position command to the servo. The signal wire's color may vary, but is normally white or yellow.

S2 Pins



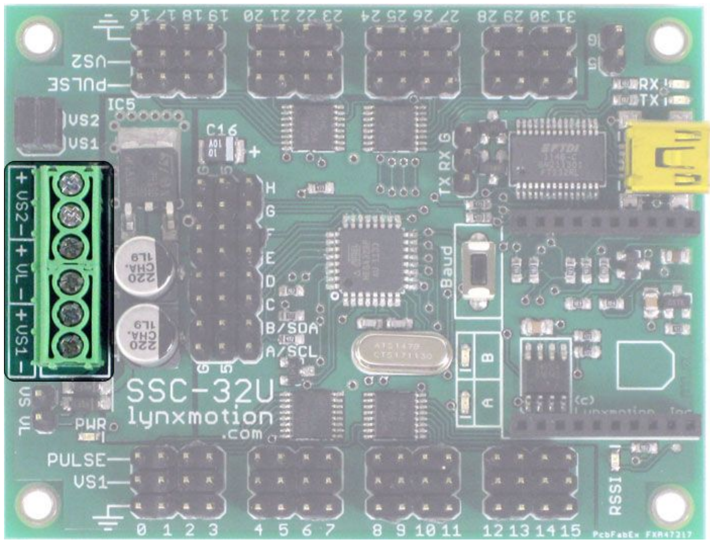
This is the second set of 16 pins. Pins 16 to 31 correspond to VS2. The outermost pin (farthest from the center of the board) is the ground pin (normally the black wire in a three pin cable). The center pin corresponds to the voltage (normally the red wire) and is marked with VS1. The last pin is the signal pin which sends the position command to the servo. The signal wire's color may vary, but is normally white or yellow.

Analog IO Pins



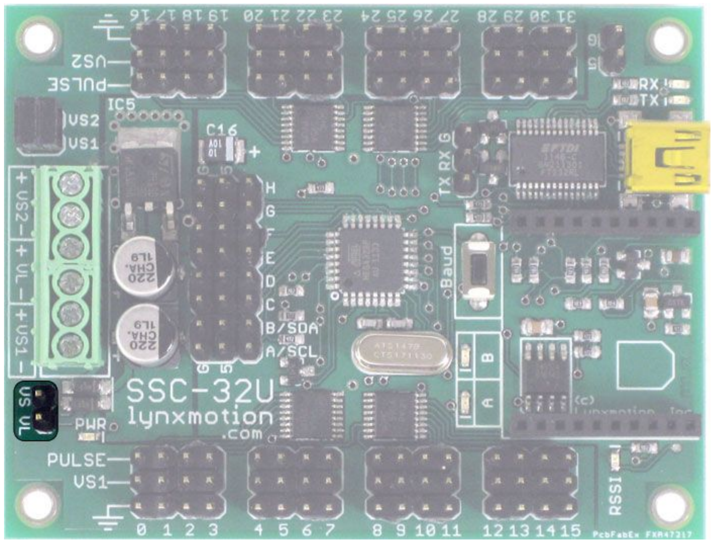
There are 8 I/O pins included on the board labeled ‘A’ to ‘H’. The row of pins on the left marked with ‘G’ correspond to ground (normally the black wire on a three pin cable). The center row of pins, marked with ‘5’ provides 5V output (normally the red wire). The last row of pins (no label) is for the signal. Two of these pins can also be used for I2C, with pin A corresponding to SCL and pin B corresponding to SDA. For I2C communication, you should also connect the GND pin, and if the SSC-32 is powering the other I2C devices, you should also connect one of the 5V pins.

Screw Terminals



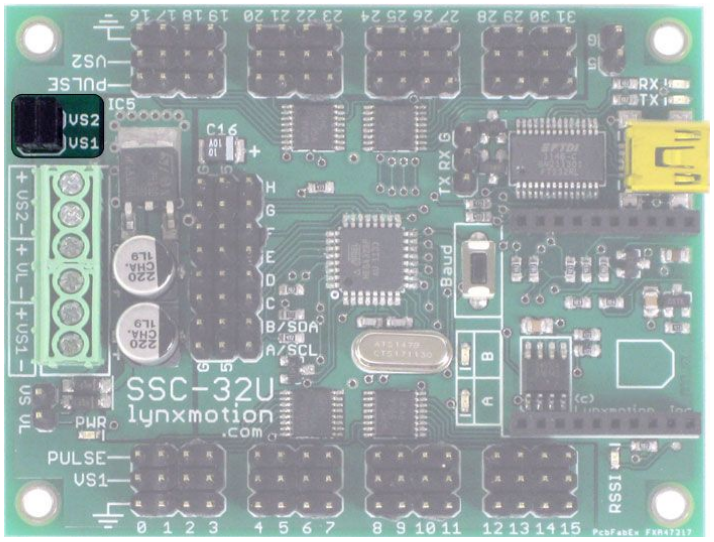
VS1 provides power directly to pins 0 to 15 (ideally 4.8V to 6V for standard RC servos)
VS2 provides power directly to pins 16 to 31 (ideally 4.8V to 6V for standard RC servos)
VL provides power to the logic controller
Be sure to connect the positive wire (usually red) to the ‘+’ and the negative wire (usually black) to ‘-’. Ensure no loose wires contact both at the same time (causing a short circuit).
Please refer to the power section of this guide for important information.

VS = VL



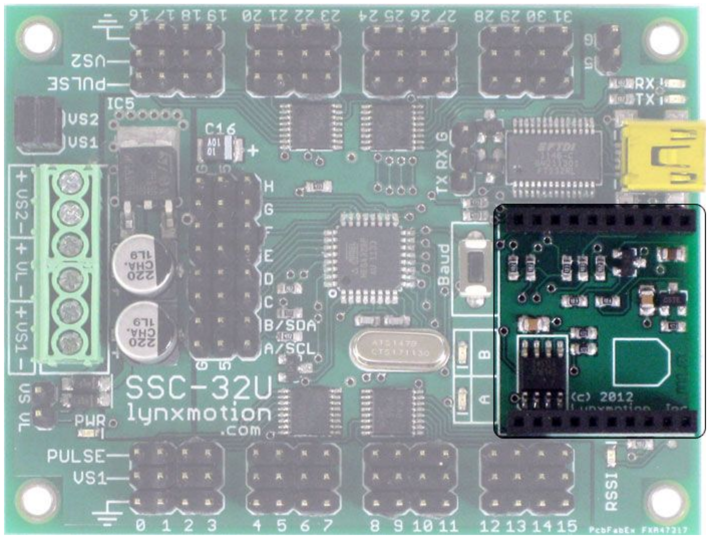
These two pins allow you to tie the VS1 terminal to VL (i.e. VL = VS). This would have the same effect as running a wire from from the VS1+ screw terminal to the VL+ screw terminal. Please refer to the power section of this guide for important information. A jumper for these pins is not included.

VS1 = VS2



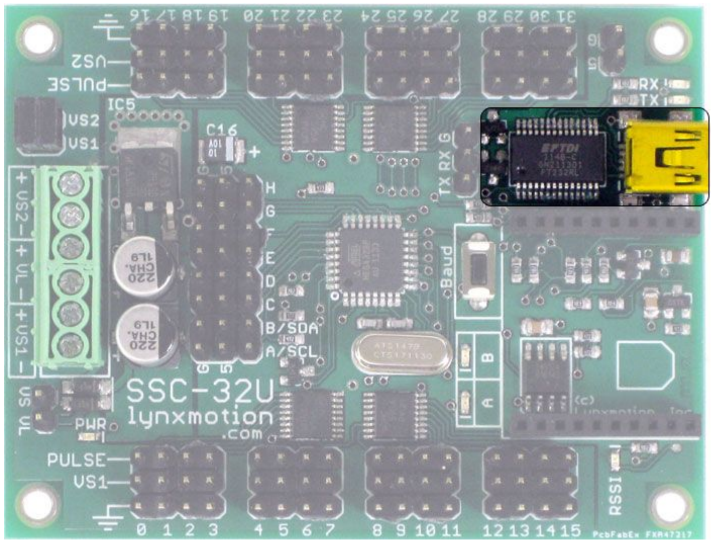
These 2x2 pins allow you to set VS1 = VS2. This would have the same effect as running a wire from the VS1+ screw terminal to the VS2+ screw terminal. There are two jumpers to handle higher current. The board is shipped with these in place and it is up to the user to remove them if needed. Please refer to the power section of this guide for important information.

XBee



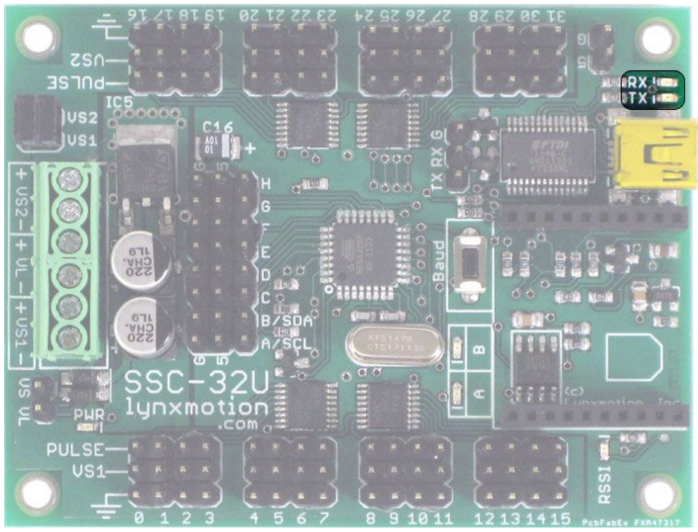
The XBee headers allow you to connect an XBee wireless module other XBee compatible devices. Take note of the image on the board for how to orient the module. Please refer to the communication / control section in this guide for more information.

USB



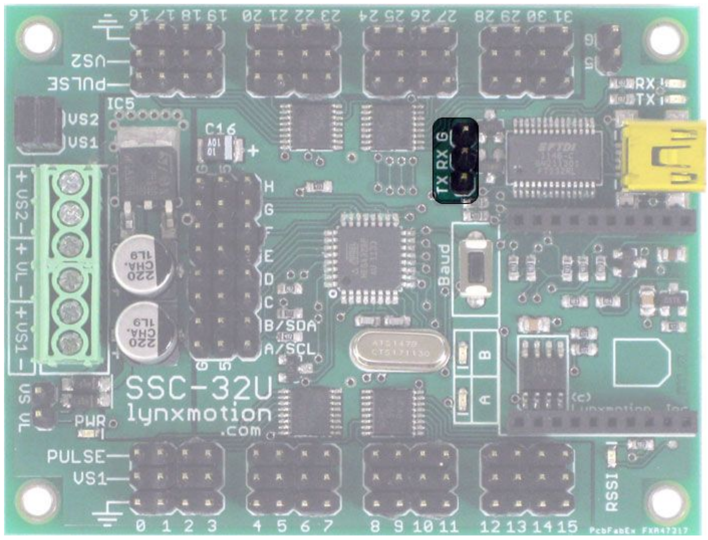
Commands can be sent to the SSC-32U's microprocessor from the computer via the onboard USB port. The microprocessor can only understand commands sent in serial (not USB) format, so an FTDI chip (the area highlighted next to the USB port) converts the USB datastream to serial and allows the board to be picked up by your computer's COM port once connected. Please refer to the drivers section below for more information.

Tx Rx



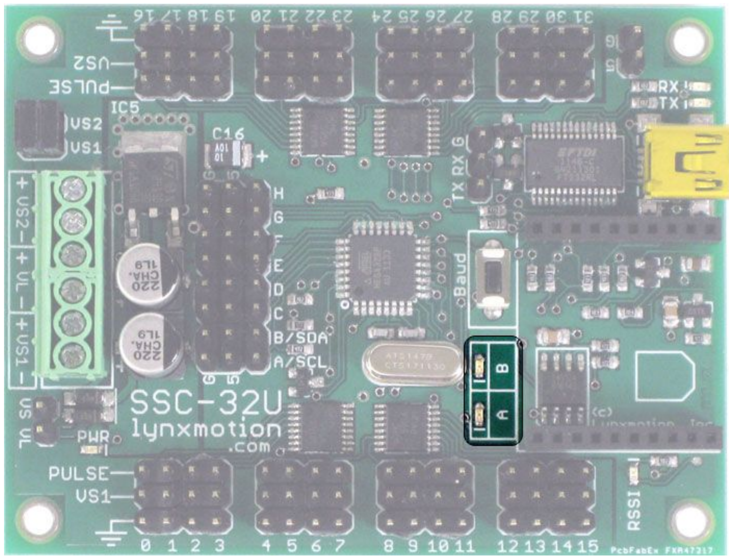
The two LEDs next to the USB port provide the user with visual feedback of when data is received (Rx) and if data is transmitted (Tx). These are only associated with communication via the USB port and not the XBee header or serial pins.

Serial



The three pins behind the FTDI chip are Tx, Rx and GND. Commands sent from the SSC-32U are done using the Tx pin while commands to be received by the SSC-32U are done via the Rx pin. These pins allow you to easily send commands to the servo controller from another microcontroller. To do so, connect the Tx pin on the microcontroller to the Rx pin on the SSC-32U, the Rx pin on the microcontroller to the Tx pin on the SSC-32 and GND to GND.

A/B LEDs

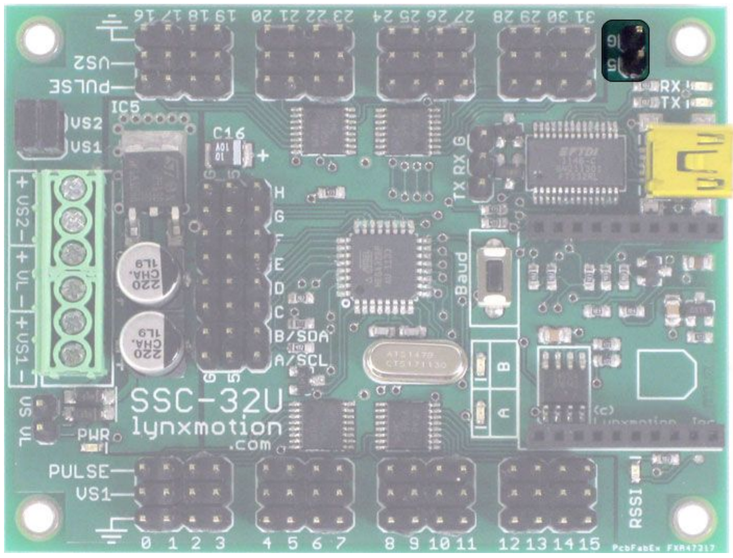


There are two distinct LEDs on the board labeled A and B When not setting the Baud rate, the LEDs indicate the following:

- Both ON at power-up before a byte is received.
- Green flashes when a valid byte is received.
- Red flashes when an invalid byte is received (framing error).

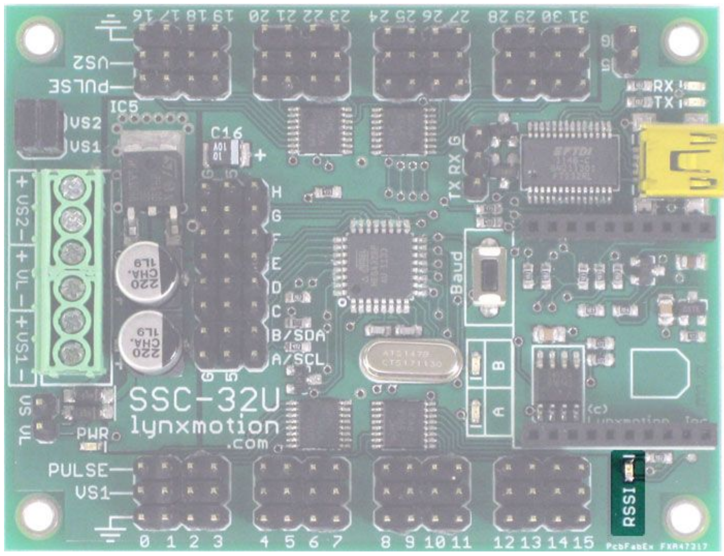
For more information, please refer to the Baud section in this guide.

5G Pins



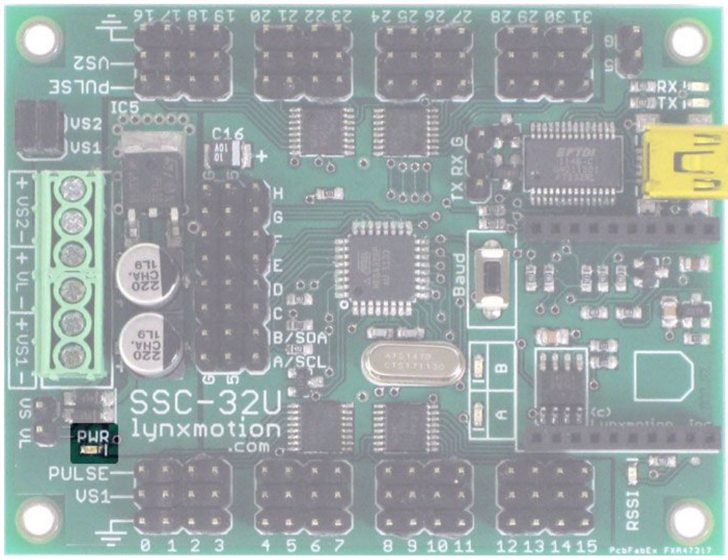
These pins are normally not needed, and pins are not provided. Should the two capacitors onboard not be sufficient for your application, a third capacitor can be added here. If you are not well versed in electronics, we suggest not touching this part of the board.

RSSI



RSSI stands for “Received Signal Strength Indication” and the LED is connected to pin 6 of the Xbee socket. Different blink intervals are used to indicate different status based on the module chosen.

PWR



The PWR LED is connected to V_logic which is the equivalent of MAX(VL, VS1) minus roughly 0.7V. It's the voltage that's then regulated to 5V for powering the logic circuits. The LED is lit when the microcontroller onboard receives adequate power.

Power Options

USB
When the board is connected to a computer via USB, the onboard USB to serial chip will be powered and your computer will be able to detect it and install the drivers. The USB does not however power the the main ATmega chip, so it must be powered separately through VS1 or VL.

VL Screw Terminal
The VL screw terminals allow for unregulated inputs to the logic voltage (the voltage used by the main processor / chip). The logic voltage is automatically selected between VL and VS1 (whichever has highest voltage). So long as VS1 is above 5.3V (not counting temporary drops in voltage), it is sufficient to power the logic voltage, and you don't need to connect anything to the VL terminals.

- Ideal: Nothing connected
- Nominal: 6-12V
- Absolute: 5.3~16V

In most situations only one battery is needed (connected to VS1), though there may be situations where it is preferable to have a battery for the logic and another for the servos:

- You are using 4.8V for your VS1, which isn't enough for VL.
- You want to cut the power to the motors but keep the logic powered.
- You want to ensure the logic is still powered even if the servo battery is depleted.

VS1 Screw Terminal
The VS1 terminal connects directly to the power and GND lines of servo pins 0 to 15. The voltage applied to VS1 should ideally correspond to the nominal voltage of the servos. For most R/C servos, this is 6V. Some “high voltage” R/C servos can operate using a 7.4V or 11.1V LiPo battery, but unless this is clearly indicated in the servo's specifications, be careful about what voltage you use. Using a 7.4V LiPo with a normal R/C servo is discouraged and using 11.1V with a normal servo will likely destroy it.

- Nominal: 6V (standard R/C servos)
- Absolute: 0~16V

VS2 Screw Terminal
The VS2 terminal is connected directly to the power and GND lines of servo pins 16 to 31.The voltage applied to VS2 should ideally correspond to the nominal voltage of the servos. For most R/C servos, this is 6V. Some “high voltage” R/C servos can operate using a 7.4V or 11.1V LiPo battery, but unless this is clearly indicated, be careful about what voltage you use.

VS1 = VS2 Jumpers

The second half of servo pins numbered 16-31 can be powered using the same power source as VS1 by leaving the two VS1=VS2 jumpers in place. There are two jumpers (rather than just one) because of the current involved. Should you want to power the line of servos connected to VS2 separately from VS1, you can remove these two jumpers. If you have the two VS1 = VS2 terminals in place, you can power EITHER VS1 or VS2, but not both. Examples of when removing these jumpers is beneficial include:

- A second battery is needed because of high current
- Second set of servos operate at a different voltage
- Want to use two separate packs for power

VL = VS

The jumper VL=VS is a way of forcing the logic voltage to use the power connected to VS1. Unless you are having unusual problems with the auto power select, this jumper should not be used. This jumper is not installed nor included in the package. Do not use the VL screw terminals if you install this jumper.

Power Considerations

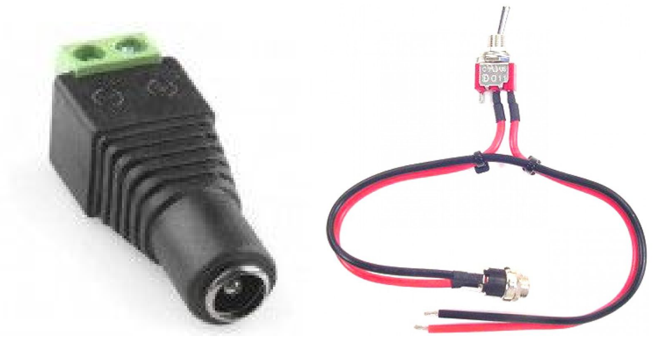
The SSC-32U board does not read the battery's voltage, and will continue to function even if a battery is being deeply discharged. It is up to the user to ensure that the pack's voltage is maintained above a certain voltage or there will be a risk of irreparably damaging the battery. Symptoms that a battery is discharged include:

- Servo speed is lower than normal
- Servos cannot maintain their position under load

Wall Adapter

Should you want to power the board using a wall adapter rather than batteries, the following should be taken into consideration:

- A wall adapter can provide an infinite supply of power, though the servos should not be operated continuously for more than around 30 minutes
- The wall adapter should be able to provide enough current (rated in amps) for all servos.
- The polarity of the barrel connector is not always evident; be sure to connect positive to positive and negative to negative / GND.
- Since the SSC-32U does not have a barrel connector, an adapter is likely needed. We suggest a 2.1mm barrel jack to screw terminal adapter, and a pair of wires, or a proper wiring harness.


Barrel Connector Input Options

Battery Chemistry & Current

Alkaline

Alkaline batteries are most commonly found as single cell 9V, AAA, AA, C and D. These batteries are not rechargeable and are usually 1.5V. You can use four 1.5V cells to make a 6V battery pack. The battery's capacity is rarely indicated on the cells, so it is difficult to gauge just how long they will last. If the battery is inexpensive, you can assume it will not last for long. The discharge rate (how much current the battery can provide either in bursts or continuously) is also rarely ever indicated. As such, if you are using several servos and notice that they do not have enough power, your batteries are likely the cause.

NiMh / NiCd

NiMh and NiCd are normally found as AA, AAA cells or as battery packs. These can be recharged and are currently the most popular type of battery for mobile robots (aside from humanoids) because of their weight and price. NiMh are preferable to NiCd because they can be recharged to full capacity every time, whereas if you recharge NiCd batteries which are only partially used, they will lose some of their capacity each time. Single cell batteries usually provide 1.2V, and as such, to make a 6V battery pack you would need 5 cells. The capacity of each cell is almost always indicated on the cell. The discharge rate tends to be 1C or 2C for most Nickel based batteries. The 'C' rating is in relation to the capacity. If a cell is rated at 2800mAh, 1C=2800mA. For example, a 1.2V 2000mAh NiMh cell rated at 1C can discharge at 2.0A continuously. Using five such batteries means you would be able to provide 6V at 2A (continuously). Battery suppliers / manufacturers create packs of several cells to provide higher voltages, and add a custom connector.

Lithium

LiPo / LiFe etc tend to be found as flat cells or rectangular packs and are normally found as multiples of 3.7V. Lithium batteries have the advantage of being lighter weight than NiMh and capable of providing significantly higher discharge rates (10C, 20C or even higher). Unfortunately for R/C servos, 3.7V is too low, whereas 7.4V is (usually) too high. We suggest using LiPo for high voltage servos only.

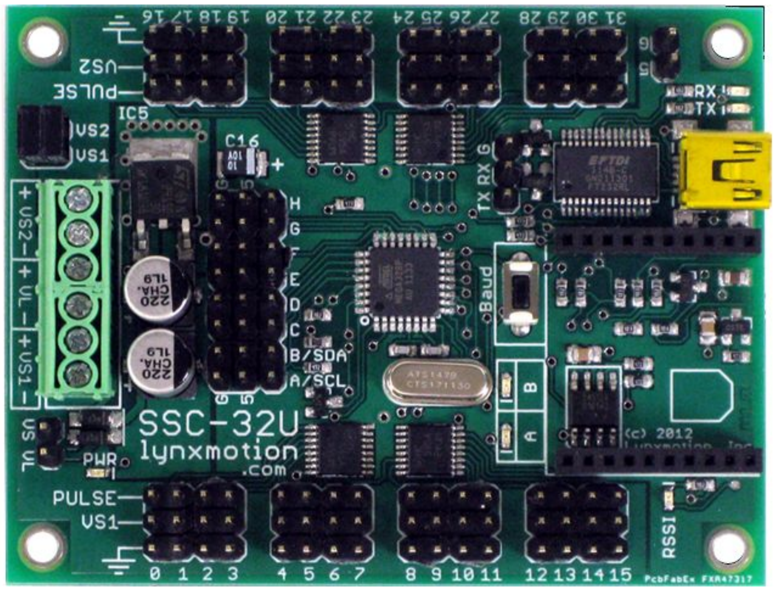
Lead Acid

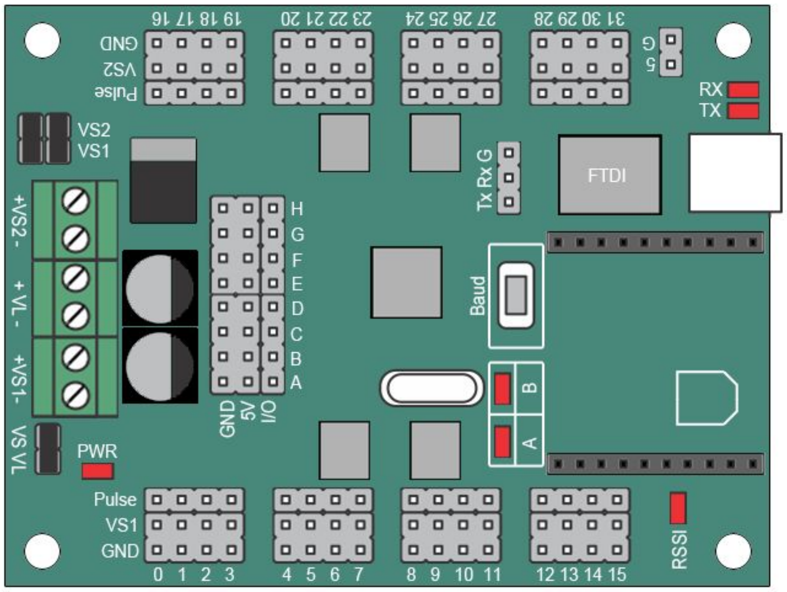
Lead acid batteries are normally 6V, 12V and 24V though many other options are possible. This is the kind of battery you would normally find in a gasoline powered vehicle. A 6V lead acid battery pack can normally provide enough current to power many R/C servos, but the battery itself will be quite heavy.

Communication / Control Options

There are three ways to communicate with the SSC-32U board: USB, TTL UART, and XBee socket. We recommend only using one, but USB will have priority if multiple are used.

The drawing below will represent the SSC-32 and will be used to show connections between the SSC-32U and other devices:

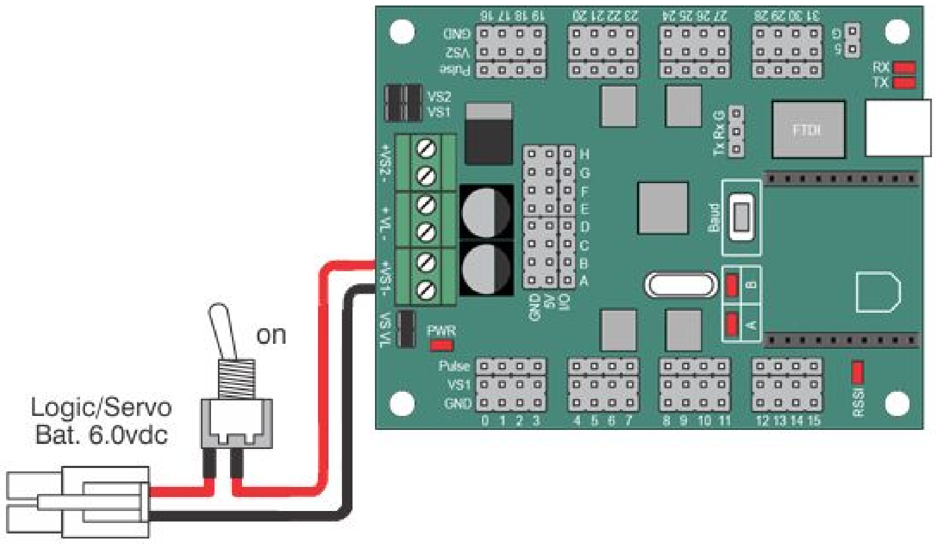

SSC-32U



Drawing of SSC-32U

USB

You can connect the board to a standard desktop or laptop computer using a normal USB cable (miniB to A). Once the USB cable is connected, your computer should automatically detect it and install the appropriate drivers to let the board appear as a COM port and allow programs on your computer to communicate with the board. If your computer does not detect the board, you should manually download and install the FTDI chip VCP drivers (based on your operating system): http://www.ftdichip.com/Drivers/VCP.htm



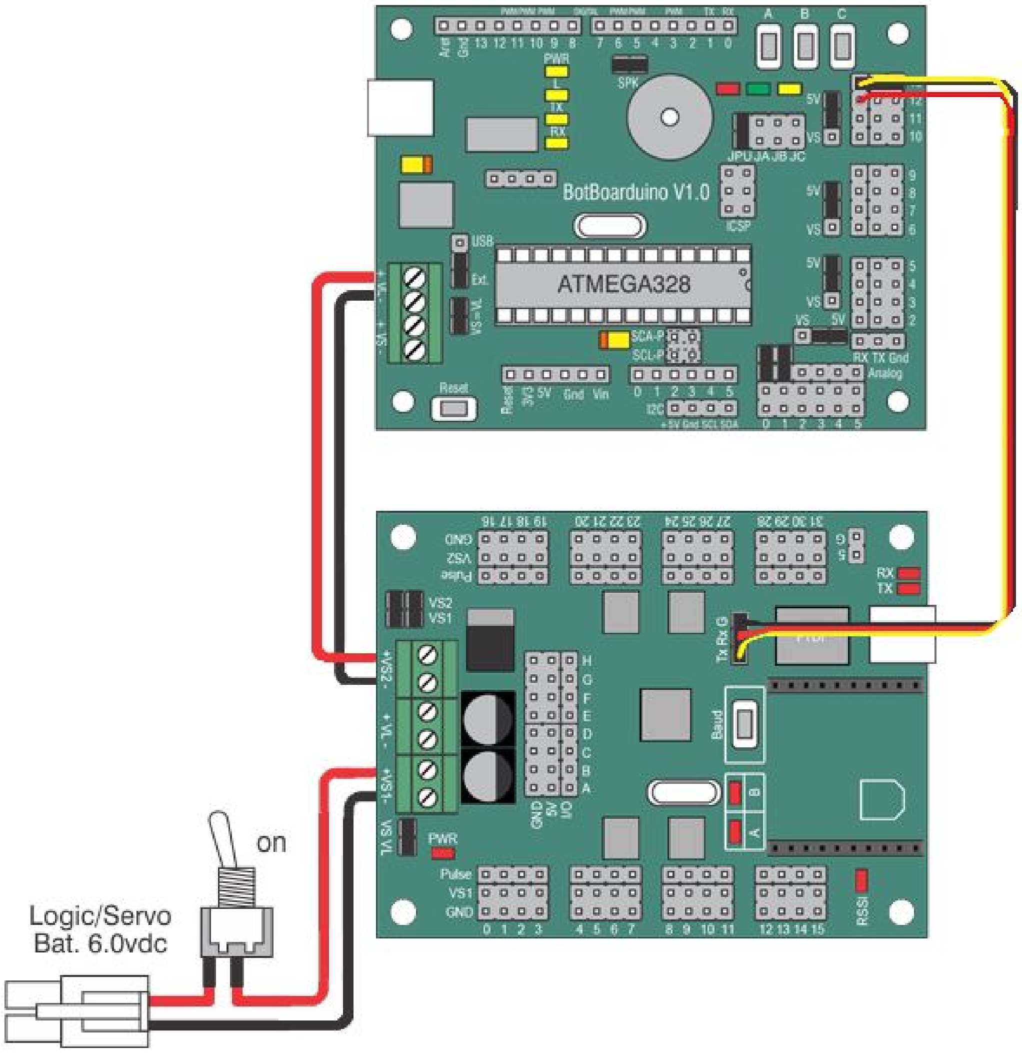
SSC-32U Power

Note that not all USB cables are identical, and some only provide power rather than power and communication. If you have issues communicating via USB with the board, first check power, then change the USB cable.

TTL UART (TX / RX / G Pins)

Commands can be sent to the SSC-32 using the serial pins on the board. There are two ways the BotBoarduino can be connected to the SSC-32

1. Direct serial connection
   - TX from the BotBoarduino to RX on the SSC-32U
   - RX from the BotBoarduino to TX on the SSC-32U
   - GND on the BotBoarduino to GND on the SSC-32U
2. Software Serial (allowing you to connect other devices to the BotBoarduino's serial pins).
   - I/O pin 13 on the BotBoarduino to SSC-32 TX (yellow)
   - I/O pin 12 on the BotBoarduino to SSC-32 RX (red)
   - GND on the BotBoarduino to GND on SSC-32 (black)

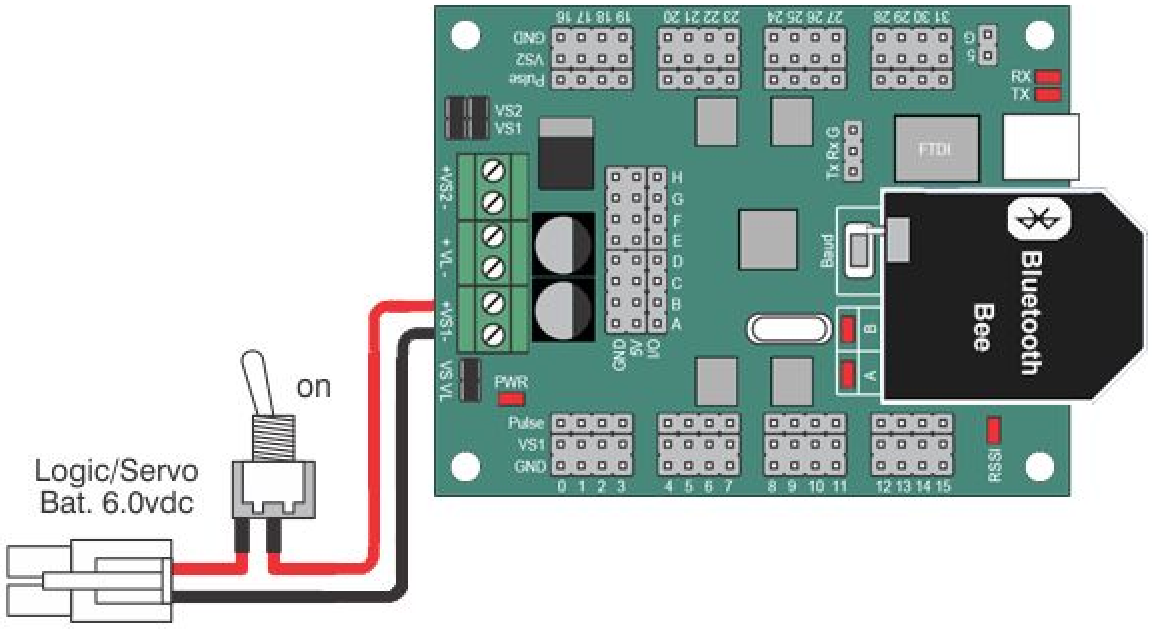


SSC-32U to BotBoarduino / Arduino

XBee headers

The XBee socket on the SSC-32U can be used with a variety of wireless devices including:

- Bluetooth modules with XBee footprint
- XBee modules
- RF modules with XBee footprint

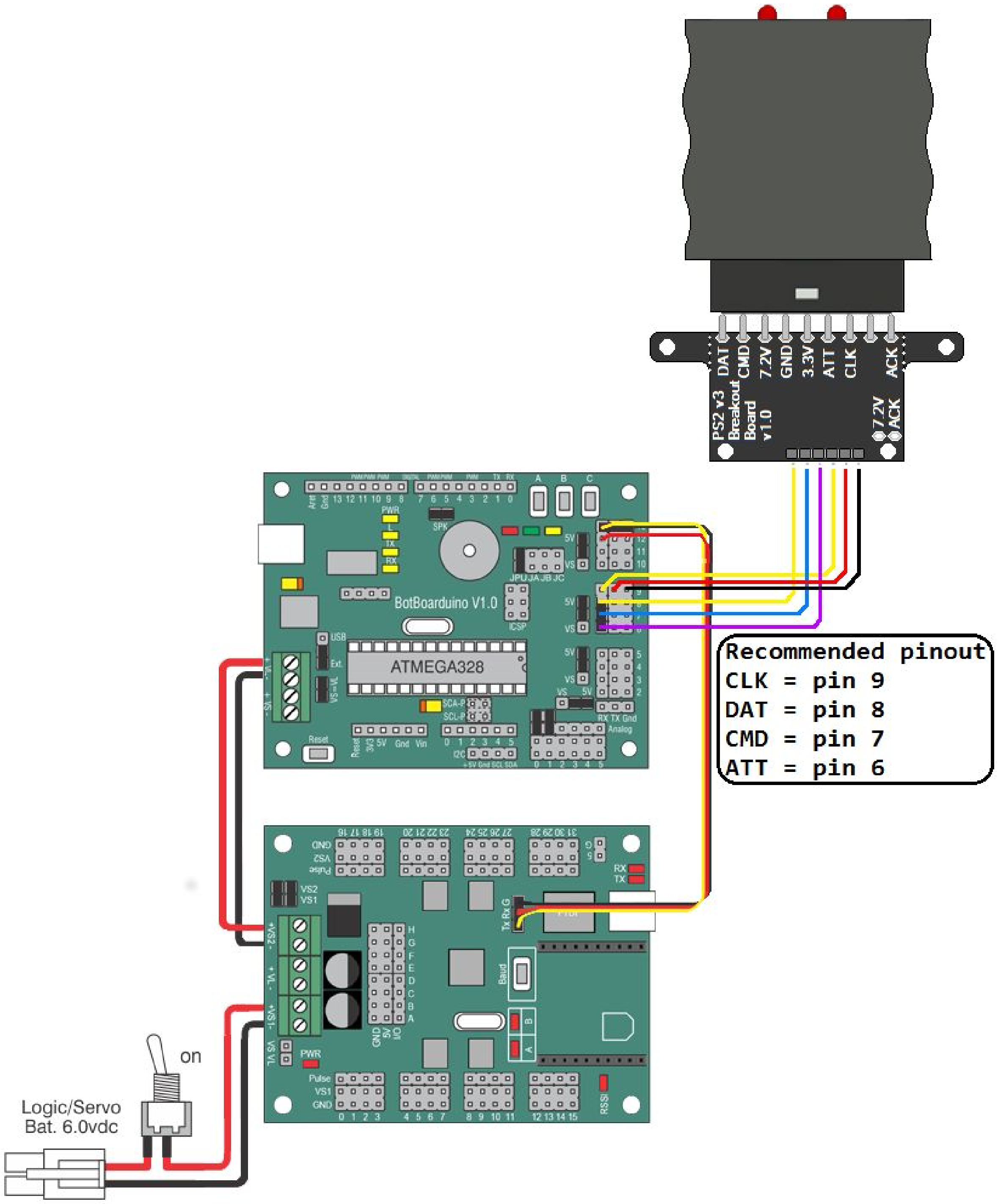


SSC-32U with Bluetooth Bee Installed

Note the orientation of the module matches that of the silkscreen on the board.

SSC-32U / BotBoarduino / PS2

The SSC-32U can be used with the BotBoarduino (and normal Arduinos), which itself is optionally connected to a PS2 controller. The Lynxmotion PS2 receiver and levelshifter is shown.



SSC-32U with BotBoarduino and Lynxmotion PS2 V3 receiver + Level Shifter

Commands

Command Types and Groups.

Single Servo Commands

In order for the SSC-32 to position a servo, it must receive a serial command in the following format. Note that the less than and greater than signs are not needed. Values in italic are optional.

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

- <ch>: pin / channel to which the servo is connected (0 to 31) in decimal
- <pw>: desired pulse width (normally 500 to 2500) in microseconds
- <*spd*>: servo movement speed in microseconds per second*
- <*time*>: time in microseconds to travel from the current position to the desired position. This affects all servos (65535 max) *
- <cr>: carriage return (ASCII 13)**

Example 1: **#5P1500S750<cr>**

This would have the servo connected to pin #5 of the SSC-32U move to position 1500 (0 degrees / centered) at a rate of 750uS per second*. 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". ASCII format is not case sensitive. Use as many bytes as required. Spaces, tabs, and line feeds are ignored.

*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 (divide 1000 by 100) to move 90°. Alternately, a speed value of 2000uS per second equates to 500mS (half a second) to move 90° (divide 1000 by 2000). Note that servos do have a maximum speed (in degrees per second), so although you may try assigning a faster speed, the servo will always be physically limited.

All SSC-32 commands must end with a carriage return character (ASCII 13). In Arduino this can be done by using the **Serial.println(); command. 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.

Example 2: **#3 P1600 T1000 <cr>**

Example 2 will move servo 3 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.

Multiple servo command (a.k.a. "Command Group")

The SSC-32U allows for multiple commands to be received in the same string. For servo position, this may look like:

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

Commands of different types cannot be mixed in the same command group. Note that the <time> function should be placed at the end of the line and is associated with the entire move, whereas the <speed> command can be associated with each servo.

Example 1: **#5 P1600 #10 P750 T2500 <cr>**

The example above will move servo 5 to position 1600, servo 10 to position 750, and it will take 2500 milliseconds (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. For example, by commanding all the legs in a walking robot with "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).

Example 2: **#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! 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 will ignore speed and time commands until the first normal command has been received.

Any move that involves more than one servo and uses either the S or T modifier is considered a Group Move, and all servos will start and stop moving at the same time. If you require moving several servos at different speeds, you must issue the commands separately.
Software Position Offset.

Servo Position Offset

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

- <ch>: channel number in decimal (0 to 31)
- <position offset>: the position offset is restricted to -100us to 100us (around 15°)
- <cr>: carriage return

The position offset command allows you to change the center position (associated with 1500us) of one or more servos via software within 15 degrees of the absolute center. The offset command only needs to be sent once to the SSC-32U servo controller at the start of the program. However when the SSC-32U is turned off it will forget the Position Offsets. Note that the servo itself is unchanged; it is only the signal sent to that particular pin(s) on the SSC-32 which will be altered.

One major uses of this function is to center / align a servo’s position perfectly with the mechanics to which it is connected. For example when assembling a servo to a bracket system, you may notice that the servo is misaligned by a few degrees with respect to the frame, whereas your code requires that it be perfectly aligned. Although correcting this issue mechanically is very difficult, the position offset command allows for fine adjustment. It's still important to build the mechanical assembly as close as possible to the desired centered position before applying the servo offset.

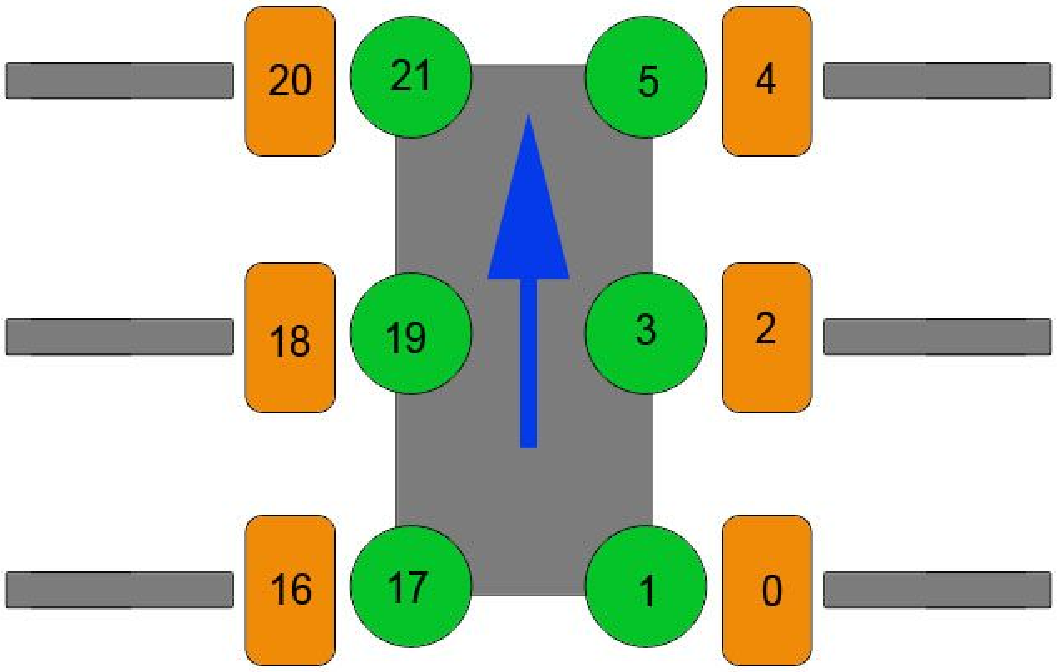
12 Servo Hexapod Sequencer

The SSC-32U includes a built-in function to easily control a 12 degree of freedom (six legs, two degrees of freedom per leg) hexapod walking robot. The function still requires serial commands to be sent from an external source (microcontroller, computer or wireless module).

The following servo channels are used for the hex sequencer (vertical / horizontal refer to the motion):

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

Hexapod Servo Pin Assignment

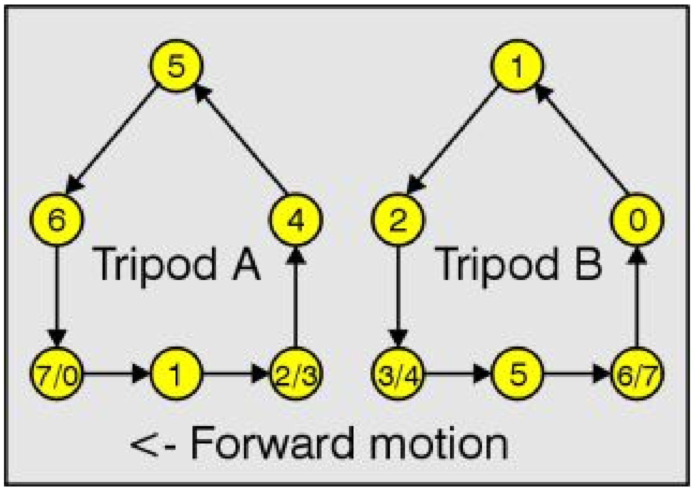


Visual Pin Assignment for Hexapod Sequencer

The walking sequence consists of 8 states, numbered 0-7. The gait uses an alternating tripod gait (three legs up, three legs down) to walking, with six possible locations of each foot.
The tripods are labeled Tripod A and Tripod B.

- Tripod A consists of {Left Front Leg, Left Rear Leg, Right Center Leg}
- Tripod B consists of {Left Center Leg, Right Front Leg, Right Rear Leg}

They are defined below:

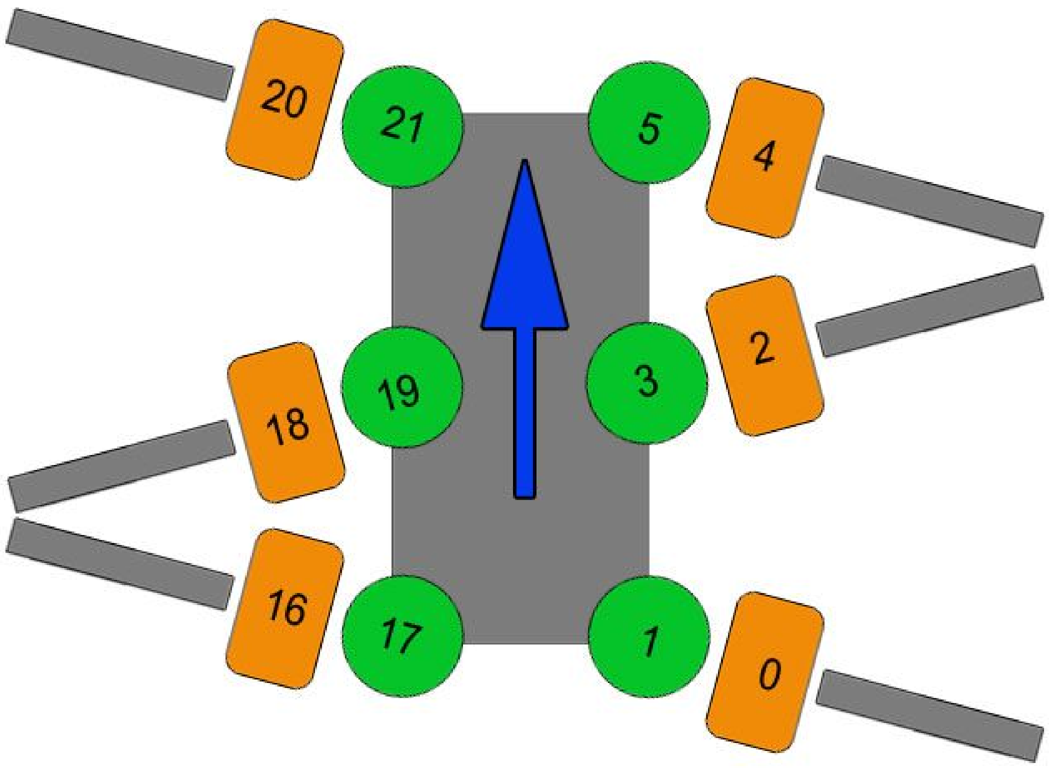


View of the leg from the side for each group of three legs

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 positions.

State	Vertical Servo (Tripod A)	Horizontal Servo (Tripod A)	Vertical Servo (Tripod B)	Horizontal Servo (Tripod B)
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

Position & Motion of Foot During Motion



Walking Gait Configuration

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; 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 values 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.

1. 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.
2. 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.
3. 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 commands.

Query Hex Sequencer State.
XQ <cr>

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).

Advanced Functions

Cancel Output

# <ch> P <pw> ….<esc>

- **<esc>**: Cancel the current action, ASCII 27

Should you wish to cancel a command, add an <esc> to the end of the line:

Discrete Output

The IO pins on the SSC-32 can be used to provide HIGH (5V) or LOW (0V) signals. Note that you should NOT use this function with standard RC servos.

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

- **<ch>:** Channel number in decimal, 0-31
- **<|v|>:** Logic level for the channel, either 'H' for High or 'L' for Low
- **<cr>:** Carriage return character, ASCII 13

The outputs on the SSC-32 come from four 8 bit shift register chips. There are four banks of 8 bit outputs as shown 0-7, 8-15, 16-23 and 24-32. The outputs can sink or source up to 20mA per output pin, but a max of 70mA per bank must be observed. The channel will go to the level indicated within 20mS of receiving the carriage return.

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
- **<cr>**: Carriage return character, ASCII 13

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 carriage return. Note that the colon (:) is required.

Example: **#3:123 <cr>**

This example will output the value 123 (decimal) to bank 3. 123 (dec) = 01111011 (bin), and bank 3 is associated with 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 at least 50uS to 5mS before the response is sent Typically the response will be started within 100uS.

Digital Input

A B C D E F AL BL CL DL EL FL<cr>

A, B, C, D, E, F are normal input reads. They read 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). As you can see, pins G and H are not included (they are only analog).

AL, BL, CL, DL, EL, FL are latching input reads. They return 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 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 simply resets the latch.

These 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.

Example: **A B E DL <cr>**

Analog Inputs

VA VB VC VD VE VF VG VH <cr>

Pins labelled A to H are analog input / output pins which can be used to read sensors, to drive low power LEDs etc.

Example: "VA VB VC VD <cr>"

VA, VB, VC, and VD read 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 is 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).

Baud
The SSC-32U is shipped with a default Baud rate of 9600. It also supports setting Baud rate using the onboard push button. To set the Baud rate:

1. Press and hold the button. At first the LEDs will glow to indicate the current Baud rate.
    a. 9600 (green)
    b. 38400 (red)
    c. 115200 (both green and red)
    d. Non-standard Baud rate (no LEDs)
2. After 2 seconds the LEDs will start to alternate, indicating you can change the Baud rate.
3. Release the button.
4. Press the button to cycle through baud rates outlined in step 1.
5. Once you have selected the Baud rate you want, do nothing; after 5 seconds the LEDs will return to normal mode and the new baud rate will be written to EEPROM.

Register R4 now holds the Baud rate. In addition to this physical way of setting the Baud rate, it can be written via the computer using the command below.
The Baud rate can be set to non-standard values if desired by writing to R4. In order to fit the Baud rates into a 16-bit value, it stores the value divided by 10; so a rate of 9600 Baud would be

stored as 960. For example, to set to 2400 Baud, issue the command R4=240. In this case, pressing the button will not light the LEDs, indicating the Baud rate is non-standard. (It will blink the LEDs briefly when pressed, just so you will know something happened.)

You can read back the current Baud rate by just entering the command "R4" followed by a carriage return.

"R4<cr>" will result in "240" if the Baud rate is 2400.

Firmware

To update the firmware, use the free SSC-32 Servo Sequencer Utility (RB_Dsp-07)

SSC-32 Registers

Register General Info

Number	**Name**	**Minimum**	**Maximum**	**Default**	**Description**
0	Enable	0	65535	0	A bit field (16 bits) that enables various features of the SSC-32.
1	Transmit Delay	0	65535	600	The delay, in microseconds, before transmitting the first byte of a response from the SSC-32.
2	Transmit Pacing	0	65535	70	The delay, in microseconds, between bytes in a response from the SSC-32.
3-31	(Reserved)	--	--	--	--
32-63	Initial Pulse Offset	-100	100	0	The initial value of the pulse offset (PO) for each servo. Register 32 corresponds to servo #0, register 33 to servo #1, etc.
64-95	Initial Pulse Width	0	65535	1500	The initial value of the pulse width for each servo. Register 64 corresponds to servo #0, register 65 to servo #1, etc. A value of 0 leaves the servo output at a continuous logic '0'; a value of 65535 leaves the servo output at a continuous logic '1'. All other values are clipped to the range 500 - 2500 microseconds.
96-255	(Reserved)	--	--	--	--

Note: Registers 0-15 are intended for global use, affecting all operation of the SSC-32; registers 32-255 are intended for individual servo channel configuration, and so are in groups of 32 registers.

Enable Register (R0) Bit Definitions

Bit	Name	Definition
15 (msb)	Global Disable	If '1', disables all of the featured controlled by the Enable register. If '0', the individual bit values will be used to enable or disable the features.
14-4	(Reserved)	--
3	Initial Pulse Width Enable	If '1', enables the Initial Pulse Width register values at startup. If '0', the default value of 0 will be used.
2	Initial Pulse Offset Enable	If '1', enables the Initial Pulse Offset register values at startup. If '0', the default value of 0 will be used.
1	TX Delay/Pacing Enable	If '1', enables the Transmit Delay and Transmit Pacing values. If '0', the default values of 600uS and 70uS will be used.
0 (lsb)	Startup String Enable	If '1', enables execution of the startup string when power is applied to the SSC-32. If '0', the startup string will not be executed.

Register Read/Write

Command	Argument	Description	Examples

Register write:
R <r> = <n> <cr>
	r = reg number, 0-255
	n = value
		Programs the value of a register. Spaces are optional around the register number and value.	R0=1023 <cr>
			R32 = -50 <cr>

Register read: R <r> <cr>	r = reg number, 0-255	Displays the value of a register, followed by a carriage return. The displayed value is in ASCII format, and is terminated with a carriage return.	R0 <cr> result: 1023<cr> R32 <cr> result: -50<cr>
Set to defaults: RDFLT <cr>	none	Sets all of the registers to their default values. When the command is complete the SSC-32 will transmit the string OK<cr>.	RDFLT <cr> result: OK<cr>

The RDFLT command may take over a second to execute. It should not be invoked while a timed move or sequence player is active. No register writes should be performed until the RDFLT is complete (as indicated by the 'ok' response).

If multiple R= commands are being sent by software, it is recommended that the software read the value of each register after it is written. This will ensure that each register write has completed before the next is started.

If an RDFLT or R= command is executing, do not power down the SSC-32 until the command has completed. To determine whether the command has completed, read a register value. Each time a register is written, the EEPROM location(s) used to store the value experience a small amount of wearout. The typical maximum number of writes is 100,000. Do not write your software to rapidly change the register values, or you might cause a permanent wearout of the EEPROM in the ATMega168 processor.

Miscellaneous Register Commands

Command	Argument	Description	Examples
STOP <n> <cr>	0-31	Immediately stops the specified servo at its current position. A space is optional before the servo number.	STOP0 <cr> STOP 31 <cr>

If the servo is part of a timed move, the other servos will continue moving and a query command will indicate that the move continues until the total time for the original move has elapsed. This is true even if *all* of the servos in the original move are stopped.

The EER and EEW commands no longer work to access internal EEPROM. They are replaced by Register Read/Write and Startup String commands. EER and EEW continue to function for external EEPROM.

Startup Strings

Command	Argument	Description / Examples
Delete characters: SSDEL <n> <cr>	0-255	Deletes <n> characters from the end of the startup string. If <n> is greater than the length of the startup string, then SSDEL deletes the entire string. SSDEL 5 <cr> - Deleted the last 5 characters of the startup string SSDEL 255 <cr> - Deletes the entire startup string
Concatenate: SSCAT <string> <cr>	Up to 100 ASCII characters	Concatenates <string> to the current startup string. The blank space immediately following "SSCAT" is ignored, but all other spaces are part of the string. The string is terminated by a carriage return, and may not contain embedded carriage returns. Commands in the startup string are terminated with a semicolon (including the last command). SSCAT #0P1000#1P2000T3000;<cr> SSCAT PL0 SQ5 SM50;<cr>
Display startup string: SS <cr>	none	Displays the entire startup string, surrounded by quotation marks and followed by a carriage return. SS <cr> result: "#0P1000#1P2000T3000;PL0 SQ5 SM50;"<cr>

The programmed startup string is executed at powerup of the SSC-32, if the Startup String Enable bit is set in the Enable register. The Startup String is executed after any register values are applied (e.g. initial pulse width). The maximum total length of the startup string is 100 ASCII characters. Any additional characters will be ignored.

The following commands should not be used in a startup string: EER, EEW, R=, SSCAT, SSDEL. The SS command may take hundreds of milliseconds to execute, depending on baud rate. It should not be invoked while a timed move or sequence player is active.
The SSCAT command may take hundreds of milliseconds to execute. It should not be invoked while a timed move or sequence player is active.

If an SSDEL or SSCAT command is executing, do not power down the SSC-32 until the command has been completed. To determine whether the command has completed, send an SS command and wait for the response. Each time the startup string is changed the EEPROM locations used to store the value experience a small amount of wearout. The typical maximum number of writes is 100,000. Do not write your software to rapidly change the startup string, or you might cause permanent wearout of the EEPROM in the ATmega168 processor.

Startup String Examples	
Command	**Result**
SSDEL 255 <cr> SS <cr>	""<cr>
SSCAT #0P2000T5000;<cr> SS <cr>	"#0P2000T5000;"<cr>
SSCAT XXXX<cr> SSC <cr>	"#0P2000T5000;XXXX"<cr>
SSDEL 4 <cr> SS <cr>	"#0P2000T5000;"<cr>
SSDEL 6 <cr> SS <cr>	"#0P2000"<cr>
SSCAT #1P1000T4000;PL0SQ5;<cr> SS <cr>	"#0P2000#1P1000T4000;PL0SQ5;"<cr>

Additional Examples	
Command	**Result**
RDFLT	Set all registers to default values
SSDEL 255	Erase the startup string
R0	Display register 0

R0=2
R1=2000
R2=1000
	Set TX delay to 2000uS and TX pacing to 1000uS. (R0=2: Bit 1 of R0 enables TX delay/pacing.)

R0=12
R32=50
R64=1000
	Set the pulse offset for servo 0 to 50 and the initial pulse width to 1000. (Bits 2 and 3 of R0 enable pulse offset and pulse width.)

R0=13
SSDEL 255
SSCAT #0P1500T5000;
	Move R0 slowly to a pulse width of 1500 at startup. Assume the initial pulse width is set as in the previous example. (Bit 0 of R0 enables the startup string.)

SS	Display the current startup string.