Peter Norberg Consulting, Inc.

Professional Solutions to Professional Problems

Ferguson, MO 63135 (314) 521-8808

Information and Instruction Manual for BC6D20NCRouter, BC6D25NCRouter and SD6DXNCRouter Firmwares on the BC6D20, BC6D25 and SD6DX series of Stepper Motor Controllers

By

Peter Norberg Consulting, Inc.

Matches Firmware Revision 5.50

Table Of Contents

Disclaimer and Revision History

All of our products are constantly undergoing upgrades and enhancements. Therefore, while this manual is accurate to the best of our knowledge as of its date of publication, it cannot be construed as a commitment that future releases will operate identically to this description. Errors may appear in the documentation; we will correct any mistakes as soon as they are discovered, and will post the corrections on the web site in a timely manner. Please refer to the specific manual for the version of the hardware and firmware that you have for the most accurate information for your product.

This manual describes board artwork BC6D20 revision 3a, with firmware BC6D20NCRouter version 5.46, board artwork BC6D25 revision 4 with firmware BC6D25NCRouter version 5.46 and board artwork SD6DX revision 5 with firmware SD6XDNCRouter version 5.46.

This manual assumes that the reader has expertise with installing and working with circuit boards, motors, and power supplies.

As a short firmware revision history key points, we have:

Version	Date	Description
5.12	June 23, 2015	Corrected TTL output error introduced in the 5.11 release. Version 5.11 could fail to set outputs to the requested values under some circumstances.
5.13	August 10, 2015	Corrected instant stop modes on asynch motors; did not operate correctly
5.24	December 27, 2015	First release of the scripting framework for all 6 axis firmwares
5.27	December 29, 2015	Adjusted startup command interpreter to allow for running a saved script
5.28	December 30, 2015	Corrected error in script save that could corrupt other saved settings if script is larger than 1K
5.29	January 1, 2016	Added CTRL-C test for scripting system and optional CTRL-C purge of input buffer. Allows immediate abort of script even if script does not monitor serial input.
5.30	January 23, 2016	Added support in the BC6D25 firmware for user control over whether a driver fault aborts all motion on the board (by default, it does…). Added capability for '-' limit input to control limits on both + and – directions (single line limit control). See the 'T' command for the new bits. Added reports of I/O direction settings and limit settings. See "-32?" and "-33?" reports.
5.32	February 14, 2016	Added control of the external ISODAC isolated voltage source, via the $J command and extensions to the 't' command. Also, corrected the standard Set Dac command ("J") to use comma-separated parameters to select the dac, instead of the current motor selection.
5.34	March 18, 2016	Adjusted vector ratio algorithm to center step points for the motors. Prior versions had step points be at last possible moment (this change is equivalent to adding 0.5 to the fractional location and then using the integer part of the result to determine when to step).
5.36	May 26, 2016	Added SENSETRANSMITEMPTY to script language, corrected implementation of script functions GoToCurrentRequestedLocationWithVectorLength, InputCharacterIfAny and PauseMotorsWithTTLRestart

5.37	May 29,2016	Added several string-related enhancements to the scripting language. Through use of SETCOMPORT, the INPUTLINE$ function now can use a list of characters as potential input terminators, and can skip a list of characters during input. A new function LASTINPUTCHARACTER reports the last character processed by the input system, so that you can check what character stopped input processing. LEFT$, RIGHT$ and MID$ allow for simple extensions to string processing.
5.38	May 31, 2016	Added DATA, RESTORE, READ$ and PRINTLINE functions to enhance application programming
5.40	June 6, 2016	Added READ command (in addition to READ$ function), which allows for reading of multiple variables in one command
5.41	June 7, 2016	Adjusted INPUTLINE$, INPUTCHARACTER and INPUTCHARACTERIFANY to actively check port 0 for pending CTRL-C characters regardless of whether port 0 is the input port, to allow scripts to be more easily aborted. Works in conjuction with bit 6 (+64) of the VERBOSEMODE command to support immediate aborts.
	June 18, 2016	Added schematic examples for connection of the SD6DX board to several of the possible external step-and-direction driver boards
5.42	September 2, 2016	Corrected error in documentation of the "IF…THEN…ELSE" clause of the scripting system. Added SetParameterList, ParameterCount and Parameter functions to the scripting language.
5.43	September 6, 2016	Corrected SetParameterList function to work correctly while inside of a GoSub
5.44	September 10, 2016	Corrected issue in Ctrl-C detection that could block it from being enabled
5.45	January 30, 2017	Correct "Block TTL Slew Bits" bit interpretation from the "T" command
	April 21, 2017	Added notes in this manual about motor selection under the scripting system
	April 25, 2017	Correct trivial typo in "!" command documentation
	April 26, 2017	Adjusted sections describing TTL I/O voltage levels for greater clarity
5.46	May 5, 2017	Added "SetCurrentCoordinates" function to scripting system
5.47	May 11, 2017	Adjusted scripting system to treat the "SetAddressMode" function as "sticky". The last state selected will remain valid (for scripts) as long as the program is loaded. It does get reset to the default (Absolute mode) on executing the "Run" command. Adjusted the AsynchGoToLocation command to be sensitive to the SetAddressMode state. Added many internal hot-links from the serial commands and their script equivalents.
5.48	May 19, 2017	Corrected error in clearing prior comma separated arguments from memory. Added "$=" to do asynch motor location assignment. Adjusted "=" to only work on vector motors. Added "AsynchSetLocation" as script function to implement the "$=" in the script system.

5.49	June 18, 2017	Added EEPromSaveUserSettings ("-123456788e" command) to the system, to allow for saving of just the user data array and the user startup commands if at all possible. This command will still save everything that can be saved (i.e., it will act like the EEPromSaveCurrentSettings command) if the current EEProm contents are invalid.
5.50	September 8, 2017	Corrected error in the scripting version of text input line. Processing of the Delete character was incorrect.

The microstep functionality in the BC6D20 and BC6D25 boards is generated by a chopper based current-sensing circuit. Although the software has a demonstrated maximum resolution of 1/16th of a full-step on the BC6D20 and 1/32nd of a full-step on the BC6D25, in practice some inexpensive stepping motors will not reliably produce unique positions to this level of precision. Mainly, the microstep feature gives you a very smooth monotonic motor action, with the capability of requesting step rates as slow as 1/16th (1/32nd) of a full step per second. We strongly suggest use of the default 1/16th of a full step microstep size; this seems to give the best performance on most motors that we tested. Most non-microstep enabled stepper motors will experience "uneven" step sizes when microstepped between their normal full step locations; however, the steps are monotonic in the correct direction, and are usually consistently located for a given position value when that position is approached from the same direction.

On the SD6DX product, you are given control of the width of the pulses that are generated as the step signals. As those widths go up, the top motion rates go down.

Product Safety Warnings

The BC6D20 and BC6D25 series of controllers can get hot enough to burn skin if touched, depending on the voltages and currents used in a given application. Care must always be taken when handling the product to avoid touching the board or its installed components, until the board has cooled down completely. Always allow adequate time for the board to "cool down" after use, and fully disconnect it from any power supply before handling it.

The board itself must not be placed near any flammable item, as it can generate significant heat. There exist several components on the bottom side of the board that can get quite hot; therefore, the board must be correctly mounted using stand-offs.

Note also that the product is not protected against static electricity. Its components can be damaged simply by touching the board when you have a "static charge" built up on your body. Such damage is not covered under either the satisfaction guarantee or the product warranty. Please be certain to safely "discharge" yourself before handling any of the boards or components.

If you attempt to use the product to drive motors that are higher current or voltage than the rated capacity of the given board, then product failure will result. It is quite possible for motors to spin out of control under some combinations of voltage or current overload. Additionally, many motors can become extremely hot during standard usage – some motors are specified to run at 90 to 100 degrees C as their steady-state temperature.

The BC6D20 and BC6D25 boards must be adequately cooled based on your intended use -- please see the separate section describing board mounting and cooling (Board mounting and cooling considerations).

The SD6DX board is not likely to ever overheat, unless you are connecting unrealistic loads to the high-current output drivers on the motor connectors.

LIFE SUPPORT POLICY

Due to the components used in the products (such as National Semiconductor Corporation, and others), Peter Norberg Consulting, Inc.'s products are not authorized for use in life support devices or systems, or in devices that can cause any form of personal injury if a failure occurred.

Note that National Semiconductor states "Life support devices or systems are devices which (a) are intended for surgical implant within the body, or (b) support or sustain life, and in whose failure to perform when properly used in accordance with instructions or use provided in the labeling, can be reasonably expected to result in a significant injury to the user". For a more detailed set of such policies, please contact National Semiconductor Corporation.

Introduction and Product Summary

*Please review the separate "**First Use**" manual before operating your stepper controller for the first time. That manual guides you through a series of tests that will allow you to get your product operating in the shortest amount of time.*

These three motor controllers from Peter Norberg Consulting, Inc., have the following general performance specifications:

	BC6D20	**BC6D25**	**SD6DX**
Unipolar Motor	No	No	N/A
Bipolar Motor	Yes	Yes	N/A
Maximum Motor supply voltage (Vm)	34V	40V	24V
Logic supply voltage (+5V)	5V	5V	5V
Quiescent current (all windings off, no fan)	70 mA	70 mA	110 mA
Maximum winding current (per motor winding, requires external fan to operate, limited duration)	2.0A	2.5A	N/A
Board size	4.90" x 2.25"	4.90" x 2.25"	4.90" x 2.25"
Dual power supply capable	Yes	Yes	Yes
USB serial	Yes	Yes	Yes
ROHS compliant	Yes	Yes	Yes
USB Ground isolation available	No	Yes	Yes

Each board can be controlled simultaneously via its TTL input lines and its USB serial interface. If the TTL inputs are used alone, then simple pan, tilt, and rate of motion are provided via input switch closures-to-ground and several connections for user-provided potentiometers; additional lines are used as limit-of-motion inputs. When operated via the serial interface, full access to the controller's extreme range of stepping rates (1 to 500,000 microsteps per second for the BC2D20 and SD6DX, 1 to 250,000 microsteps per second for the BC6D25), slope rates (1 to 1,000,000,000 microsteps per second per second), and various motor motion rules are provided. The BC6D20 boards have a theoretical microstep resolution of 1/16th of a full step, while the BC6D25 boards have a theoretical microstep resolution of 1/32nd of a full step. Both use a constant-torque algorithm when operating in microstep mode.

Short Feature Summary

- A full scripting system is available for motion control without a computer.
- One through six stepper motors may be independently controlled at one time.
- Any combination of motors may be treated as 'vector' synchronous motors, while the other motors may simultaneously as asynchronous independent motors.
- For the BC6D20 and BC6D25, each motor must be Bipolar, but may be of differing current requirements.
- For the BC6D20, each motor may draw up to 2.0 amps/winding. For the BC6D25, each motor may draw up to 2.5 amps/winding. Note that an external cooling fan **must be used** when your motor draw exceeds 0.9 amps per winding or the motor supply exceeds 19 volts.
- Supports a separate backlash setting for each motor
- Limit switches may optionally be used to automatically request motion stop of any motor in either direction.
- Rates of 0.001 to 500,000 microsteps per second are supported on the BC6D20 and SD6DX units, while 0.001 to 250,000 microsteps per second are supported on the BC6D25. When operated in NC mode, the maximum rate is 250,000 microsteps per second for all products.
- Step rates are changed by linearly ramping the rates. The rate of change is independently programmed for each motor, and can be from 1 to > 1,600,000,000 microsteps per second per second.
- For the BC6D20, all motor coordinates and rates are always expressed in programmable microunits of up to 1/16th step. On the BC6D25 all motor coordinates and rates are always expressed in programmable microunits of up to 1/32nd step. On the SD6DX, the units are steps.
- Motor coordinates are maintained as 32 bit signed values, and thus have a range of -2,147,483,647 through +2,147,483,647.
- When operating in vector mode, the code automatically scales motor motions between the selected motors, so that the 'fastest' motor operates at the target rate which you specify, and so that all motors arrive at their target destinations at effectively the same time. You may also specify a 'virtual' motor which is used as the rate specifier; all other motors get their rates scaled (up and down) to allow that 'virtual' motor to operate at the requested rates.
- Permits 'chaining' of goto-vectors, so that you don't have to stop between each line
- A TTL "busy" signal is available, which can be used to see if the motors are still moving. This information is also available from the serial connection.
- Complete control of the motors, including total monitoring of current conditions, is available through the USB serial connection.
- Runs off of a single user-provided 6.5 to 15 volt DC power supply, or two supplies (5V or USB power for the logic circuits and 7.5-34V for the motors).
- Any number of motors may be run off of one serial line through use of the included daisy-chaining "serial routing" capabilities. Note that the automatic "serial routing" system is disabled when the controller is operating in its scripting mode of operation.
- 'Slew' inputs may be arbitrarily reprogrammed as generic TTL inputs and outputs
- Several of the I/O pins may be individually reprogrammed as 12 bit A/D inputs, TTL inputs, or as TTL outputs.
- ESD protection to most of the TTL inputs and outputs is provided, as is optional powering of the on-board logic (not the motors) via the USB 5 volt signal.
- Supports connection of up to two phase encoders (by reassigning several of the TTL I/O lines as phase encoder inputs)
- On the SD6DX unit, 2 extra outputs are provided per motor to allow automatic selection of motor currents in external driver boards when motors are moving or idle.

Asynchronous and Synchronous modes of motion

The firmware simultaneously supports both asynchronous and synchronous modes of operation. The "T" command in conjunction with the motor selection "m" command is used to control whether a give motor is connected to the asynchronous control system or the synchronous control system.

The asynchronous mode allows for fully independent simultaneous motion for each motor, with optional control of motion via TTL inputs (the SLEW inputs) and optional control of rates of motion via voltage inputs. The motion is purely based on the most recent command to each given motor; no support is provided for buffered motion sequencing.

The synchronous mode allows for treatment of any set of the 6 motors as vector motion engines. It supports a 100 element queue of pending motion requests, with each motion request being a vectored scaled move of all motors to their requested next location, with rate scaling done between each motor so that all motors reach their target location simultaneously. This motion is also called 'NC' motion (for Numerically Controlled), and is the style of motion needed to operate NC milling machines or any system that requires perfect rate scaling between motors.

Asynchronous motion operations

When operated asynchronously, motors selected by the "m" command respond to the following actions as configured by the "T" command:

- TTL input slew control, via the SLEW input switches associated with the motor
- Limit-input based stop requests, with either high or low limit inputs inducing the stop action
- Voltage based rate control, with "$r" used to control interpretation of the input voltages into rate actions

Additionally, serial-command based motion may be performed. The related actions include:

- "m" – Selects which motors are to respond to the commands
- "$K" - Sets the start/stop rate of motion
- "$P" – Sets the ramp rate for moving from one rate to the next
- "$R" – Sets the target rate of motion for any new or current motion events
- "$G" – Tells the motor to "goto" a given absolute location
- "$M" – Either memorizes the current location, or goes to the memorized location
- "$S" – Tells the motor to "seek" a given number of steps relative to the current location (or to just spin in a given direction)

For example, to tell motor X to go to location 300 at a rate of 875 microsteps per second, you may send the following commands:

```
2m      - Select motor X
875$R   - Set the target rate to 875
300$G   - Tell motor X to go to location 300
```

Synchronous (Vector) motion operations

When operated synchronously (as vectors), the set of motors that comprise the vector respond to vector oriented commands. Usually, these commands are buffered into a 100 element queue, and are executed strictly sequentially. Some commands (most notably the 'arc' related actions) generate a series of queued requests, and will abort generation of the complete sequence if a new command that needs to be queued is requested before the entire series of queued requests has been generated (i.e., only part of the arc might be generated). As long as your application maintains communications synchronization with the controller and respects the queue status that is reported by the system, this will not be an issue for you.

As an optimization, some of the vector commands optionally support comma-separated parameters. This reduces the communication overhead in setting up for the given commands, since only the full command generates a handshake as opposed to each parameter requiring handshake synchronization. On most of the commands that accept comma-separated parameters, you may precede the sequence with one semicolon-separated parameter, which is normally used to specify the rate for that command.

Since most of the vector motion commands involve operations on a queue, there are several commands which are used to report on the current status of the queue. In all of these status commands, a report is given that tells you the amount of space that is left in the queue and the status tells you what the controller is doing.

For vector motion operation, there is an internal pseudo-motor that is used to control motion ramping and core rates of actions for the 'single' vector action. All of the other motors (that are selected as 'vector/synchronous' motors via the "T" command) are 'slaved' to that pseudo-motor, with their actual steps being an exact percentage of the pseudo-motor's steps, with the ratio being the length of motion for the given motor to the length of motion of the pseudo-motor. The "K", "P" and "R" commands are used to set the rates of the pseudo-motor, while the "G" and "g" commands actually define the vector and queue the motion.

One of the key elements of NC control is how each vector connects to the next in terms of rate of motion. By default (if you do not do any of the special actions that are available for overriding this behavior), each vector is fully disjoint from the next vector. The code starts each motion at the start/stop (K) rate, ramps at the ramp rate (P) to the target rate (R) and then reverses the process at the end of the vector. If you need one vector to 'join' with the next without this slow-down effect, there are several methods available. Note that each vector segment may always be specified with its own target rate; the code will automatically ramp the rate up and down as needed to try to attain your request (which may or may not be possible, depending on your ramp rates and your length of motion).

1. Use of the "M" multi-vector chain mode command with a fixed parameter of '-1' states "All of the following vectors are to be merged into one trapezoidal ramp event, until you either run out of vectors or until you issue the '0M' call". This is the easiest method of control, since you start your sequence with a '-1M', issue your various 'G'/'g' based vector requests, and complete the motion with a '0M'. If you are unable to 'feed the queue' rapidly enough to keep the motion operating smoothly, the firmware automatically slows motion down when it finds that it does not have enough queued vectors left to avoid a sudden stop due to running out of data, and it speeds it back up as soon as there are enough motion events to allow it to do so.

2. Alternatively, you can provide 'M' with a positive value that is the sum of all of the pseudo-motor vector lengths for your entire motion event. In this case, the code will trust that you will feed the queue correctly, and will operate the vector system to give you one trapezoidal motion over the requested number of steps of the pseudo-motor. In this mode, if you do not feed the queue rapidly enough, you will get sudden-stop events if the queue becomes empty.

3. Finally, you can mostly bypass the standard trapezoidal rate of motion of the vector system, and force your own rates for each vector. You may use the 'k' command to force the current start/rate for the incoming vector to be whatever

you want, and you specify a negative rate (such as '-800R') to force the ramp mode to be purely 'ramp to target rate', without the automatic clean slow-down at the end of the motion.

You may also inject non-motion events in between the vectors, noting that this will cause a 20 microsecond delay in motion for each event that is inserted. The TTL output commands may be queued (by using a leading '-' before the parameter for the output), and you may also inject generic timed (p) or ttl-input-based (q) pauses into the motion system. The pauses are instant (they occur exactly at the point in the queue where they are seen); it is important that you set up your vectors to have the motors at the start/stop rate by the time that a queued pause is executed, since the motion stop will otherwise be sudden.

One additional set of motions are the arc/spiral system. In that class of motion, you specify which pair of motors are to be treated as if they were an X,Y pen plotter onto which an arc (or spiral) is to be drawn, and you specify target locations for all of the other motors. For the motors that are to directly participate in the arc, you specify the arc parameters (radius (and second radius if a spiral is to be 'drawn'), center or start location, start and delta angle relative to that location, and how many segments (or each chord length) to draw, and the code then will queue up all of the chained vectors needed to draw your complete arc or spiral. The extra motors will move linearly to their targets based on even motion for each line segment drawn.

Vector Motion Configuration

Vector Motor Motion Control

Arc generation

Default Firmware Configuration: Power-On (and reset) Defaults

All of the default settings on the board (such as microstep size, ramp rates and so on) may be reprogrammed by you to be remembered after a power cycle event (see the 'e' command for EEProm operations). However, all of those features have a 'factory default' value which is used whenever an EEProm reset is done or when a 'reboot using factory default' command is given ("-1!").

- **BC6D20: 1!** – Set the microstep size to 1/16 of a step per logical step.
- **BC6D25: 2!** – Set the microstep size to 1/16 of a step per logical step.
- **SD6DX: 2!** – Set the pulse width to 2 microseconds
- **63m** – Select all motors for the following actions
- **9600b** – Set the communications baud rate to 9600 baud
- **0**= – Reset all motors to be at location 0
- **BC6D20 and BC6D25: 500H** – Set motors to 0.5 Amps when moving
- **SD6DX: 1H** – O1 high, O2 low when moving
- **SD6DX: 0J**: both DACS are set to 0 after reset
- **80K** – Set the vector mode "Stop OK" rate to 80 microsteps/second
- **80$K** – Set the asynchronous mode "Stop OK" rate to 80 microsteps/second
- **0o** – Set scale factor for specifying motor current to milliamp mode
- **8000P** – Set the rate of changing the vector speed to 8000 microsteps/second/second
- **8000$P** – Set the rate of changing the motor speed to 8000 microsteps/second/second
- **800R** – Set the target run rate for the vector to 800 microsteps/second
- **800$R** – Set the target run rate for the motor to 800 microsteps/second
- **0T** – Enable all limit switch detection, set all motors synchronous (vector mode). On the SD6DX, this also makes control of the O1/O2 outputs be automatic.
- **0t** – Set all I/O ports to our defined default settings
- **1V** – Set <CR><LF> sent at start of new command, no transmission delay time
- **BC6D20 and BC6D25: 0N** – Windings are off when idle
- **SD6DX: 2N** – O1 low, O2 high when idle

Cooling Requirements

On the BC6D20 and BC6D25, if you are operating motors that require more than 900 mA of current per winding, or if your motor voltage exceeds 19 volts, then you normally must provide for fan-based cooling of the board. We suggest a total of at least 20 CFM (in our dual fan configuration, we provide two 13 CFM fans for this), directed downward towards the board so that both the voltage regulator and the driver chips are in the direct path of the airflow. You may need to provide fan cooling even if the current is less than 900 mA, depending on the ventilation you provide in your enclosure.

If you intend to leave the power windings on (via the variations of the 'N' command), or if you are not providing for any air flow in your enclosure (i.e., you are using a sealed box), then you are very likely to have to cool the board, even if the current is under 900 mA/winding.

Please note: If a fan is needed, then the board must be mounted with at least 1/4 inch clearance underneath. If your motor current draw is to be 1.5 amps/winding or higher, then the clearance must be 1/2 inch (or more), or there must be an active side-flow of air to keep the underside of the board cool.

Please look at the last section of the manual, Install The Fan Assembly, for instructions on mounting our 5 volt fan assembly directly on the board.

The SD6DX unit normally will not require a fan.

Board Signal Jumpers

Most of the I/O signals on the board (all of the limit and slew inputs and the reset input) have fixed non-adjustable resistors installed for pullup to the I/O power rail (selected

Several of the I/O signals on the board may be used as both TTL I/O and as analog inputs. When used as analog, they should not have pullup resistors installed; when used as TTL I/O, they usually should have pullup resistors installed. This set of jumpers gives you control over use of the pullup resistors.

RDP – Pullup for the RDY signal

NXP – Pullup for the NXY signal

IO2P – Pullup for the IO2 signal

SI2P – Pullup for the SI2 signal

SO2P – If present, pullup for the SO2 signal

SOP – On some beta artworks, pullup for the SO signal

Normally, all of these signals EXCEPT for SI2P are in the position that enables the pullup. Remove the jumper (or position it so that just one pin is connected to anything) to remove the pullup. Note that SI2P should not be installed when using TTL-serial I/O.

Board Power Option Jumpers

The boards have several jumpers that relate to power distribution.

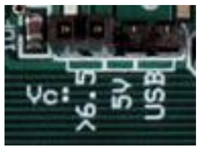
Vc: POWER Jumper – How internal board 5 volts is generated

The "Vc" jumper defines the voltage that you are providing to the +Vc regulated DC power input to the board. You can provide exactly 5 volts (which is then routed directly to the internal power bus of the board), a regulated DC voltage between 6.5 and 15 volts (which the board then regulates down to the 5 volts required for the local logic power bus), or you may power the board logic off of your USB system. This jumper has three positions:

'>6.5' – Defines use of an external 6.5 to 15 volt power supply
'5V' – Defines use of an external 5 volt power supply
'USB' – Defines use of the USB bus as the voltage source for the system 5 volt bus

If you operate in the '5V' mode, please be aware that **you can trivially destroy the board** by supplying a voltage which is not well-regulated 5V DC. **At anything above 5.5 volts, the board will definitely be destroyed quite quickly**, while anything above 5 volts will cause the microprocessor to run hotter (and possibly overheat). If the voltage ever drops to 4.3 volts or less, the microprocessor will act as if it has had a power failure – it will lose all settings and perform a restart action.

On each board, the jumper appears as follows:



The 'USB' position enables powering the logic portion of the board from the USB 5 volt system. When the jumper is in the USB position, power is applied to most of the board logic only while the computer is 'awake' – that is to say, if the computer is turned off, asleep, or in some hibernation mode, then the board will act as if no power is present. For artworks that have the feature, you must also install the "USBGND" jumper if you are selecting the USB power option.

Note that the board draws about 80-150 mA (depending on use of the TTL signals) without the fan assembly, and as much as 350 mA with the fan assembly. This means that, for the USB-powered mode to work, the connection must be a high-power connection (either directly to the computer, or via a good powered hub).

This jumper is normally shipped in the **>6.5** position, with the expectation that you will be using a 6.5 to 15 volt supply to provide power via the Vc/GND power inputs.

Pullup Voltage (or TTL I/O Voltage) – Voltage level used for TTL I/O

The Pullup Voltage jumper (also called the "TTL I/O Voltage" on the SD6DX) controls the voltage levels presented at the TTL I/O signals. This may be either 5 volts or 3.3 volts.

SD6DX only: Output Voltage – Controls the step-and-direction output voltage source

The SD6DX also has an "Output Voltage" jumper. Normally, this is left in the 5V position, which means that the signals on the motor outputs are all pulled up to the board +5 volt. For oddball step-and-direction drivers that need higher voltages, this may be moved to the +Vc position, which means that all of the signals get pulled up to the voltage present on the +Vc power input.

USBGND – Connects the USB ground to the board ground

The release artworks for the SD6DX and BC6D25 units have an additional “USBGND” jumper. If the board is to have its logic powered off of the USB system (via the ‘Vc’ jumper set to the ‘USB’ position), this jumper must be installed. In all other power configurations, it is optional. You may wish to operate with this jumper removed in order to provide for ground isolation between your computer and the controller board, to simplify power grounding systems.

Analog Input Signals – Accept 0 to 3.3 volts

Three of the input signals (NXT, RDY and IO2) may be configured through use of the ‘t’ and 'a' commands to act as analog inputs. When they are configured this way, they accept **0** to **3.3** volt positive voltages, which map into A/D readings of 0 to 4095 (respectively).

When those pins are used as analog, the associated pullup jumpers (NXP, RDP and IO2P) must be removed, or board damage may result!

The input system requires less than 2.5K of input impedance: this means that your signal sources need to be somewhat ‘hard’ (higher current), otherwise you may get more noise on the input than you would prefer.

You may also find that you get better results by placing a small capacitor (0.01 to 1 uF) between the selected input (such as NXT) and GND, as a simple filter for the input. If you do this, you will get better results having the capacitor being close to the actual input connector than if the capacitor is at your voltage source, since it will help to reduce induced noise picked up by your wires running from your signal source to the board.

You use the ‘a’ command to read the signals, as is described here.

TTL Signal Notes

All external connections are done via labeled terminal block connections and one USB serial port. All of the motor and power connections are on one side of the board, while all of the TTL I/O and logic connections are spread around the remaining 3 sides of the board.

TTL Input Voltage Levels

Internally, all TTL input signals use 3.3 volt logic levels (but are actually "tolerant" of up to 5 volt levels). This means that a logic "0" is generated at any time that the input voltage drops to <= 1 volt, and a logic "1" is generated when the input voltage is above about 2.3 volts. Signals in between those voltages are in a 'grey' area due to potential noise on the power supply.

Externally, the TTL voltage levels are controlled by the Pullup Voltage jumper setting, as described in the manual section "Pullup Voltage (or TTL I/O Voltage) – Voltage level used for TTL I/O". The TTL voltage I/O level may either be set to 5 volts or 3.3 volts.

The LIM and SLEW TTL inputs are tied to the pullup (3.3 or 5 volt) voltage via 10K resistors (depending on the Pullup Voltage jumper setting, see Pullup Option Jumpers). This permits you to use switch-closure-to-board-ground as your method of generating a "0" to the board, with the "1" being generated by opening the circuit. You may rely on this resistance value as being valid for a current-based driver.

The extended TTL signals of RDY, NXT, IO2, SO2 and SI2 use 1K pull-ups to the Pullup Voltage, most of which may be disabled by removing their associated jumpers. The lower resistance allows a higher current to be generated when those signals are changed by you to being outputs.

On some artworks the TTL signal SO2 uses a fixed 1K pullup; on those artworks, that pullup may not be disabled by you.

Note that when the SLEW TTL signals are treated as outputs, they are still pulled up by the 10K resistors. This means that they can only drive low-draw circuits – at 5 volts, there is only 0.5 mA available for highs (i.e., the 10 K pullup), but they are pulled down with up to a 2 ma driver.

For the RDY, NXT, IO2 and SI2 signals, there is 5 mA available (at 5 volts) when high, and the signals are pulled down with 8 mA drivers.

Input Limit Sensors, lines LW- to LZ+ and LA- to LB+

Lines LW- through LZ+ and LA- through LB+ are used by the software to request that the motors begin to stop moving when they reach a hardware-defined positional limit. Enabled by default at power on, the firmware also supports the 'T' command, which may be optionally used to enable or disable any combination of these switches.

The connections are:

Signal	Limit Sensed
LW-	-W
LW+	+W
LX-	-X
LX+	+X
LY-	-Y
LY+	+Y
LZ-	-Z
LZ+	+Z
LA-	-A
LA+	+A
LB-	-B
LB+	+B

The connections may be implemented as momentary switch closures to ground; on the connector, a ground pin is available on each TTL I/O connector that contains limit inputs. They are fully 3.3 and 5 volt TTL compatible; therefore driving them from some voltage-based detection circuit (such as an LED sensor that generates 0 to 3.3 or 5 volt outputs) will work. The lines are "pulled up" to +3.3 or +5V (depending on the Pullup Voltage jumper) with a 10K resistor, external to the microcontroller.

The stop requested by a limit switch normally is "soft"; that is to say, the motor will start ramping down to a stop once the limit is reached – it will **not** stop instantly at the limit point. You may use the 'T' command to reconfigure this behavior to be an instant stop, if that is needed.

Under the 'soft' stop, if a very slow ramp rate is selected (such as changing the speed at only 1 microstep per second per second), it can take a very large number of steps to stop in extreme circumstances. It is quite important to know the distance (in microsteps) between limit switch actuation and the hard mechanical limit of each motorized axis, and to select the rate of stepping ("R"), rate of changing rates (the slope, "P"), and the stop rate ("K") appropriately.

As the most extreme example possible:

- if for some insane reason the motor is currently running at its maximum rate of 500,000 microsteps per second,
- and the allowed rate of change of speed is 1 microstep per second per second,
- and the stop rate was set to 1 microstep per second,
- then the total time to stop would be 500,000 seconds (138 hours -- groan!), with a distance of ½ v^2, or ½ (500,000)^2, or 125,000,000,000 microsteps.
- Note that this same amount of time would have been needed to get up to the 500,000 rate to begin with…

Therefore, it is strongly recommended that, if limit switch operation is to be used, these extremes be avoided. By default, the standard rate of change is initialized to 8000 microsteps/second/second, with the stop rate being set to 80 microsteps/second.

Use of the "!" emergency reset command will cause an immediate stop of the motor, regardless of any other actions or settings in the system. ***Please be aware that, in some designs, damage to gear systems can result when such a sudden stop occurs. Use this feature with care!***

If you configure the system for 'instant stop' using the 't' command, ***please be aware that, in some designs, damage to gear systems can result when such a sudden stop occurs. Use this feature with care!***

Motor Slew Control: A- to B+, W- to Z+, IO2, NXT, RDY

Lines A- through B+, W- through Z+, IO2, NXT and RDY may used to control stepping of the motors, and the rate of steps when the associated motor is configured as an asynchrous motor (see the 'T' command). When an associated motor is set up as being synchronous (i.e., is associated with vector motion), these signals are available as generic TTL I/O lines. The 'u' command may be used to set their I/O direction, while the 'D' (set ttl values), 'C' (set bits high) and 'U' (set bits low) commands set the values when they are outputs.

The RDY output signal is normally configured by you to indicate that motor motion is still being requested on at least one of the motors. When HIGH, then all motion is stopped. When LOW, at least one motor is still moving.

Alternatively, the 'T' and 't' commands allow specification of use of any combination of the IO2, NXT and RDY signals as analog rate controls for any combination of asynchronous motors. 'T' tells each motor whether it is synchronous or asynchronous and which analog input (if any) is used to control its rate, while 't' is used to select how RDY and NXT are configured (as analog inputs, instant stop actions, or 'board idle' reports).

Analog Input Signals

Three of the input signals (IO2, NXT and RDY) may be configured through use of the 't' command to act as analog inputs. When they are configured this way, they accept 0 to 3.3 volt positive voltages, which map into A/D readings of 0 to 4095 (respectively).

The input system requires less than 2.5K of input impedance: this means that your signal sources need to be somewhat 'hard' (higher current), otherwise you may get more noise on the input than you would prefer.

You may also find that you get better results by placing a small capacitor (0.01 to 1 uF) between the selected input (such as NXT) and GND, as a simple filter for the input. If you do this, you will get better results having the capacitor being close to the actual input connector than if the capacitor is at your voltage source, since it will help to reduce induced noise picked up by your wires running from your signal source to the board.

You use the 'a' command to read the signals, as is described here

USB Driver Installation Under Windows

Our boards use a USB driver chip for communications with your hosing computer. FTDI (http://www.ftdichip.com) provides drivers for operation of these chips under Windows^{tm}, Linux, and Mac/OS. Our installation disk includes **modified** copies of their Windows^{tm} drivers; our boards normally use a unique ID code which prevents them from being recognized by the default FTDI drivers (although you may special-order them using the default codes). All Linux and Mac/OS customers must download their drivers directly from the http://www.ftdichip.com site, as we have no support capability for those platforms. They will also have to special-order the boards from us such that we configure them to respond to the standard FTDI drivers. Look for the drivers and documentation that relate to their FT232RL device.

A short summary of the installation of the drivers under Windows follows. For installation under Linux or on the Mac, please refer to the FTDI documentation available from their web site.

Base Driver Installation Under Windows XP, Vista and Windows 7

The installation of the USB drivers has been heavily streamlined and simplified. Normally, the installation is done as one of the intermediate steps of installing the documentation package (from the SetUpStepperBoard.exe installation application provided on our CD or as a download from our web site). If you elect to not do the installation at that time, the actual current version of the installer is installed in your programs menu, under:

Start->Programs->StepperBoard->USB Support Tools->Install FTDIStepperBoard USB Drivers

This tool requires that you are logged on as an administrator: it will not correctly install the drivers if you are a ‘standard user’. The process normally is:

1. If you have not already done so, log on as an administrator to your computer. If you are not logged on as an administrator, then the following procedure will be blocked as soon as you attempt to install the drivers!
2. Make sure that none of our boards are plugged in.
3. Run the installer to completion, and exit any remaining “completion” panels.
4. Insert the small square end of the A-B USB cable into the stepper motor control board. Insert the large flat end into a free USB port into your computer. **Please make certain that the cable fully “snaps” into our connector – this can take a noticeable amount of force**. If it is not properly “seated”, then Windows will not correctly recognize the board.
5. Once the installation process completes, the code will automatically add a new “COM” serial port which is “attached” to the board when it is plugged into the same USB port on your computer. *The system will automatically add a new COM port each time you attach a new board to any other USB port on your computer or hub. It may also create a new COM port if you receive a repaired board back from us (if we have had to replace the USB driver chip).*
6. Note that it can take quite a while for Windows to fully activate the COM port whenever you insert the cable. For example, on a Windows 7 Professional installation on a 3.3 GHz 4-core computer, it frequently takes 15 seconds or longer for the COM port to become active. On other versions of Windows and on slower machines it can take much longer.

Initial testing of any BC6D20, BC6D25 or SD6DX board after driver installation – TestSerialPorts

The easiest way to test the board (and to identify which COM port is being used for board communications) is to run our "TestSerialPorts" application (found under 'StepperBoard' on your 'Start' menu). This application will scan all of the potential COM ports on your system (from COM1 through COM255), and will identify every port that has a connected StepperBoard product powered and attached.

The test assumes that the board is correctly configured to 'talk' to the com port. The board must be correctly powered for the TestSerialPorts application to be able to locate the board.

When TestSerialPorts starts, simply press the "Scan Serial Ports" button (you may safely ignore the other buttons). The application will then perform its scan, and will identify every COM port on your system. It will also identify the baud rate for each connected board.

If TestSerialPorts does not locate your board, please contact us for additional tests to perform. Remember that the board must be connected to your computer and powered on (and the FTDI USB drivers must be correctly installed, if the USB version of the product is being used) for TestSerialPorts to be able to locate the board).

Please note that the TestSerialPorts application will locate our board even if you have not adjusted the default COM port properties, as described in the prior section. It will **NOT** locate our board if the board has not been correctly powered! It also will NOT locate the board if you have placed the board into its scripting mode of operation (which means that you have already "talked" to it….)

Adjusting Default COM port properties for best operation

Once the system has created the COM port for the board, you may need to change the system defaults to match the requirements of our motor controllers. If you are using the StepperBoard class library (either the Visual Basic 6.0 version or the Visual Studio .NET version, you do NOT need to change the COM port defaults – the current release of those packages use the FTD2XX.DLL library from FTDI that gives the code access to the required functionality).

If you installed from the 'FtdiStepperBoard' subdirectory or via the installation packaged provided with the board, then these changes will normally have been done for you. Otherwise, you may need to perform the following procedure:

1. Under Windows 2000 or XP, go to your "system properties" page. Do this by
    a. Right-click on your "System" icon
    b. Select "Properties"
    c. Select "Hardware devices" (it might just be called "Hardware")
    d. Select "Device Manager"
2. Under Windows Vista, 7, 8 or 10, log in as an administrator, and then get to your "device manager" page by:
    a. Go to your 'Start' menu, and click on the 'Computer' button
    b. On the ribbon that appears at the top of the resulting window, click on "System Properties"
    c. On the task pane on the left of the new window, click on "Device Manager"
    d. The system will ask for your permission to continue. Press the "Continue" button.
3. Look under "Ports (COM and LPT)", and select the COM port that you just added (it will normally be the highest-numbered port on the system, such as "COM6"), and edit its properties. Note that the 'TestSerialPorts' application (described on the prior page) will have identified this COM port for you as part of its report.
4. Reset the default communication rate to:
    a. 9600 Baud,
    b. No Parity,
    c. 1 Stop Bit,
    d. 8 Data Bits,
    e. No Handshake
5. Select the "Advanced Properties" page, and set the:
    a. Read and Write buffer sizes to 64 (from their default of 4096).
    b. Latency Timer to 1 millisecond
    c. Minimum Read Timeout to 0
    d. Minimum Write Timeout to 0
    e. Serial Enumerator to checked
    f. Serial Printer to unchecked
    g. Cancel If Power Off to unchecked
    h. Event On Surprise Removal to unchecked
    i. Set RTS On Close to unchecked

(that is to say, only the Serial Enumerator is checked in the set of check boxes on the display)

Serial Operation

The TTL based serial control of the system allows for full access to all internal features of the system. It operates at almost any legal baud rate from 2400 to 115,200, no parity, and 1 stop bit. Any command may be directed to the W, X, Y, Z, A, B or all motors; thus, each motor is fully independently controlled. Note that you should wait about ¼ second after power on or reset to send new commands to the controller; the system does some initialization processing which can cause it to miss serial characters during this "wake up" period. You may use the 'b' command to select the baud rate: it will accept almost any rate, although only a few will be 'perfect'. The suggested rates to use would be 9600, 19200, 38400, or 115200.

Actual vector control of the stepper motors is performed synchronously for all motors. Motion commands (i.e., goto actions) are queued in the controller, and executed serially. Motors that are configured as asynchronous will execute their commands as soon as they are presented to them.

Serial input either defines a new current value, or executes a command. The current value usually remains unchanged between commands; therefore, the same value may be sent to multiple commands, by merely specifying the value, then the list of commands. For example,

1000X

would mean "set the next X target location to 1000"

2000G

would mean "queue the current W, X, Y, Z, A, B request, with a vector length for the purposes of rate management of 2000".

Some commands explicitly require a unique value to be specified each time that they are used, to avoid incorrect operation of the board. For example, 'accidentally' requesting a baud rate change via a 'b' with no parameter would give you an unusable board (until it is repowered) if the code did not detect that no unique parameter was given on the command.

The firmware actually recognizes and responds to each new command once the stop bit of the character has been received. This means that the command starts being processed just after completion of the character bit stream. In most designs, this will not be a problem; however, since all commands issue an '*' upon completion, and they can also (by default) issue a <CR><LF> pair before starting, it is quite possible to start receiving data pertaining to the command before the receive buffer has been initialized! In microprocessor, non-buffering designs (such as with the Parallax, Inc.^{tm} Basic Stamp ^{tm} series of boards), this can be a significant issue. This firmware provides a configurable option in the 'V' command which adjusts the timing. If enabled, the code will "send" a byte of no-data upon receipt of a new command character. This really means that the first data bit of a response to a command will not occur until at least 7-8 bit intervals after completion of transmission of the stop bit of that command (about 750 uSeconds at 9600 baud); for the Basic Stamp^{tm} this is quite sufficient for it to switch from send mode to receive mode.

Standard command serial sequence

For most operation of the controller (the exception occurring when you tell the controller to operate using its Basic-like scripting system, at which point all bets are off), all commands are sent out as a (set of) values, followed by a single command. Each command that is noted by the controller will at least be acknowledged by an "*" character; there may be other characters sent by the controller before the "*" which depend upon the command and on other board settings.

One command sequence is thus:

- Send a sequence of digits (possibly comma and semicolon separated), which define the parameters for the command.
    - These consist of numeric characters "0" to "9", "-", "+", ".", and optional parameter separators of "," and/or ";".
- Send a single (or double) character command (the double character commands all start with the "$" character
- Optionally receive a <CR><LF> (carriage-return, line-feed) response (depending on the "V" command settings)
- Receive any data response from the command (such as a comma separated list when performing a generic "?" query). This list may have multiple <CR><LF> sequences embedded within it.
- Receive an "*" character to note that key processing of the command has completed (on motor motion, this normally means that the motion has been queued or started; only a few commands actually wait for motion to complete before sending the "*" response)

The StepperBoard class library function "StringWriteCommand" simplifies serial processing

The low level portion of the StepperBoard class libraries make use of the above "standard" to simplify sending of command strings into one standardized tool ("StringWriteCommand"). An inspection of that code will show that it:

- Sends all characters from the set of "0123456789-+<>$.,;" and any character whose ASCII value is at or above that of the "{" character without expecting any response (note that the "<", ">", "$"characters and those at or above "{" are special cases that change the board states without sending a response)
- Sends the single command character
- Accumulates the text response from the controller until the "*" portion is seen
    - The code includes automatic switching between two standard (programmable) timeout values for the time that it takes to receive the "*", to allow code to abort if an expected response is not seen in the appropriate time frame

Routing Serial to 'Child' Boards

The BC6D20, BC6D25 and SD6DX series of boards support optional "Serial Routing" of data to and from other boards via the "SI2" and "SO2" TTL-Serial connectors. This permits "daisy-chaining" of the boards, so that more than 6 motors may be easily operated off of one serial port. Additionally, the firmware supports "binary" serial transmissions – that is to say, it allows communication with a "child" board which includes non-visible characters, thus permitting communication with other products not supplied by Peter Norberg Consulting, Inc.

Please note that serial routing (using the commands described in this section) does NOT work when a script is running. When a script is in control, all serial I/O is under control of the script, not of the default communications system.

Connecting your 'Child Board' – How to use SI2 and SO2

The SI2 and SO2 signals on the BC6D20/SD6DX are normally used as the TTL-Serial connections to the child board. These are NOT RS232-level signals; instead, they are standard TTL signals, 0 to 5 (or 3.3, if you have selected 3.3 volt operation) volts in value. The signals are:

- SI2: TTL-Serial Input received from child board
- SO2: TTL-Serial Output sent to child board

You also need to make certain that both boards have the same ground potential: they both need to have their logic supplies run off of the same power supply. Ground should be done via a "star" ground, which means that all grounds are tied to a common point (NOT "daisy-chained"!).

When operated in TTL-Serial mode, the "SI2P" jumper on the board MUST be removed, and the SO2P jumper must be installed if it is present!

For example, to "daisy chain" two BC6D20 boards, your connections would be described as follows:

```
Parent board:
        USB connected to the computer
                (or its SI/SO lines connected to some other TTL-Serial source)

Child board:
        Its SI/SO lines connected to the "parent" SO2/SI2 lines (as shown below)

Wiring:
        Parent SO2 → Child SI
        Parent SI2 → Child SO
```

Please note that SI2 of the parent is connected to SO of the child, and SO2 of the parent is connected to SI of the child!

Routing the Serial Data

At power on, the board is configured to directly interpret and execute all incoming serial data, and to leave the SI2/SO2 serial echo lines idle. This mode means that the board is under control of the serial data stream from the host. There are two methods of exiting this mode, and switching to serial-routing operation (wherein data is routed from the serial port to and from the SI2/SO2 pins as needed).

One method, the "Binary Route" technique, allows single-character entry into route mode, and is the preferred method when controlling only one child board or when controlling a child board which is not provided by Peter Norberg Consulting, Inc. (and therefore does not match our normal communications protocol).

The other method, the "SerRoute compatible" technique, allows the board to operate as both a router and a motor controller within the rules as shown below.

'<' – Start Binary Route

The '<' character immediately starts routing all serial data to and from the child board. Except for the route command characters ('<', '>', '{', '}') and the ESC character ('\'), all characters from the serial port are sent unchanged to the SO2 child serial port without any other interpretation by the firmware.

This mode is the preferred method when operating any board as a child that is not a Peter Norberg Consulting, Inc. product. It permits transmission of binary data, and avoids any issues of sending spurious route-related commands (such as the '{' "start route selection" character).

Please note that, when routing in the binary route mode, you must precede certain characters with the escape character '\' in order to prevent the firmware from interpreting those characters as route commands. This means that you should precede any of '<', '>', '{', '}', or '\' with a '\' ("escape it"), so that it will be passed on unchanged to the child board.

'>' – Stop Routing

The '>' character immediately stops any routing which is taking place. This is normally used to exit the routing mode, and return to "talking" to the main board.

Note that if the '>' character is preceded by the '\' escape character, the '>' is just sent on to the child (assuming that routing is enabled), and route mode is not canceled.

'{xxx}' – Select SerRoute compatible Route Mode

This sequence allows routing using a method which is compatible with the SerRoute firmware. The 'xxx' values are used to select which "route" is to be enabled, while the '{' and '}' are used to bracket the route address. Please see the 'SerRoute' manual for a full explanation of the routing rules: from the point of view of BC6D20, however, there are only 3 important SerRoute-compatible route modes, as selected by the first character in the 'xxx' value.

x	**Description**
Empty	If just '{}' is sent, then the current board is selected for all serial. This is exactly equivalent to receipt of the '>' character, above.
'0'-'8'	All serial data is routed to pin SO2, and all serial data seen on pin SI2 is routed back to the host.
'9'	Both this board and the child board receive all following serial data, while the serial data sent back to the host is strictly that from this board. Used to execute commands simultaneously on both boards.

In reality, when interpreting the '{xxx}' style of input, the code operates as follows:

1. When the '{' is seen, the code sets a flag for 'the next character tells us how to route all of the following characters'.
2. When the first 'x' character is seen, if it is not '}' or '0'-'9', the route command is cancelled (illegal sequence), and the 'x' character is interpreted as a new board command. I.e., '{g' is interpreted as 'g' (although technically speaking, this form of command is not proper).
3. If the first 'x' character is '}', then routing is terminated. All following characters are treated as normal data to the board, and no further activity occurs relative to the SI2/SO2 ports (aside from completing any pending serial transmission).
4. If the first 'x' character is '0' to '8', then all future serial data is routed to the child board via the SO2 and SI2 lines. The '{' "route selection" character is passed on to the child, to allow for nested route control.

5. If the first ‘x’ character is ‘9’, then (as with ‘0’ to ‘8’) all future serial data is replicated to the SI2 output line. However, it is also executed locally, thus placing the board into the mode of ‘broadcast’. The ‘{‘ “route selection” character is passed on to the child, to allow for nested route control.

For example, if we have 3 BC6D20 boards with BC6D20NCRouter firmware “daisy chained” such that:

- the first board “talks” via its SI and SO lines to the host (possibly using the USBToTTL or RS232ToTTL level conversion board, if needed)
- the second board has its SI line connected to the first board’s SO2 line, and its SO line to the first board’s SI2 line, and
- the third board has its SI line connected to the second board’s SO2 line, and its SO line to the second board’s SI2 line, giving us:

First Board

→ Second Board

→ Third Board

We would address the first board with:

{}

The second with:

{0}

The third with:

{00}

And all three at once with:

{99}

Serial Commands

The serial commands for the system are described in the following sections. The code is case-sensitive – 'G' and 'g' have different meanings! ***Please be aware that any time any new input character is received, any pending output (such as the standard "*" response to a prior command, or the more complex output from a report) is normally cancelled.*** This behavior may be controlled through the 'V' command.

The two modes of motion control – vector and asynchronous

The firmware supports two methods of driving each motor. They may either be run as vector systems (wherein the selected group of motors has their rates of motion scaled such that you get the equivalent of 'straight line motion' if you were to connect and n-dimensional pen plotter to the system) or asynchronous systems (independent motion for the motor; no connection with any other motor). Through use of the "T" command, you tell each motor how it is to be treated. From then on, that motor will only accept commands that affect its mode of operation.

The 'asynchronous only' commands all are two-letter commands starting with a '$' (i.e., '$G' is an asynchronous goto request); the only vector commands that start with a '$' are the arc commands "$A" and "$a", which were needed due to overlap with the vector 'A' motor location specification command 'A' and the read a/d command 'a'.

The 'vector motion' commands operate on the group of motors that have been configured (the 'T' command) as being vector motors. Each motion request is entered into a queue of pending actions, and there are control strategies in place that define how rate ramping is performed within each vector and between vectors.

Serial Command Quick Summary

Most of the commands may be preceded with a number, to set the value for the command. If no value is given, then the last value seen is used if the command is not a 'configuration' command. The command will be ignored if there is no data for the command and it is a configuration request (i.e., 't' alone would be ignored, since no data is given and it is a configuration command). Some commands support comma separated arguments as their list of parameters (such as "23,6#" to set the current encoder count values for both encoders).

All commands are case sensitive!

General Commands

0-9, +, - – Generate a new VALUE as the parameter for all FOLLOWING commands
b – Select the communications baud rate
L – Latch Report: Report current latches, reset latches to 0
n – Reset the current queued element counter
V – Verbose mode command synchronization
! – RESET – all values cleared, all motors set to "free", redefine microstep. Duplicates Power-On Conditions!
? – Report status
# – Define current encoder values
[- Specify startup command string

Motor Control Configuration

e – Write EEProm data
F - Set motor backlash
f - set rate scale factor
j – Set motor microstep size on the BC6D20 and BC6D25 units
H – Specify motor current when running
N – Set motor current when idle
o – Specify scale factor to use to interpret H and N values (motor current units)
T – limiT switch control

Motor Selection

m – Select motors to be accessed on following commands

Asynchronous Motor Motion

$G – Goto a requested absolute location
$S – Slew to a requested relative location
$M – Mark current location or goto marked location
$K – Set the asynchronous "Start/Stop" rate
$P – sloPe (number of steps/second that rate may change)
$R – Set run Rate target speed for selected motor(s)
$r – Set adc scaling for asynchronous analog rate control

Vector Motion Configuration

E – Report queue element data
n –reset sequential ID associated with the next queue element
K –Set the vector mode "Start/Stop" rate
P – sloPe (number of steps/second that rate may change)
R – Set run Rate target speed for selected motor(s)

Vector Motor Motion Control

G – Queue a request to Go to absolute position w, x, y, z, a, b
g – Queue a request to Go to relative position w, x, y, z, a, b
M – Set multi-vector chained motion mode
h – report current motion status
I – Wait for motors fully 'Idle'
i - wait for motion command queue space
k - If in non-trapezoidal mode, force instantaneous current rate to be the requested value.
p – insert timed pause into queue

0-9, +, - – Generate a new VALUE as the parameter for all FOLLOWING commands

Possible combinations:

"-" alone – Set '-' seen, set no value yet: used on SLEW -

"+" alone – Clear '-' seen, set no value yet: used on SLEW+

-n: Value is treated as -n

n: Value is treated as +n

+n: Value is treated as +n

Examples:

-$S – Start slew in '-' direction on the current motor

-10$S – Slew back 10 steps on the current motor

$A – Draw an Arc of the requested radius (DrawArc and DrawSpiral)

This vector-mode command draws an "arc" (actually, a series of connected straight lines) whose radius is that specified, and whose other parameters were specified by the current state of the system. The complete arc is specified by the X, Y, d, c, S, $a and $A commands:

- The most recent two W, X, Y, Z, A and B commands normally set the CENTER point of the arc. If you provide a negative radius (i.e., "-1000" as opposed to "1000"), then the pending X, Y value becomes the first point on the arc as opposed to the center point (with the center being calculated based on the remaining parameters).
- '**d**' defines the signed "arc delta", where the angle units are degrees. The sign defines the direction for the drawing of the arc. If a value of 0 is specified, then the action is to 'just go to the start arc location'. Note that decimal values are allowed (i.e., '22.2355d' is a valid request for a delta angle of 22.2355 degrees).
- '**c**' defines either the count of line segments drawn for the arc or the segment length for each line segment. '0' means 'just go to the start arc location'. A positive value specifies the segment count, while a negative value specifies the segment length.
- '**S**' specifies the beginning angle (again, in units of degrees, with decimal values allowed)
- **'$a'** specifies the ending radius of the circle (thus allowing a spiral to be generated if it does not match the '**$A**' value). If not specified, the **$a** value is set to match the **$A** value.
- '**$A**' specifies the radius of the circle, and actually draws the line

The code moves all six motors as part of this command. The last two motors specified via the "W, X, Y, Z, A, B" commands are used as the virtual X and Y axes for the arc generation; the remaining motors motion gets scaled so that their motion completes at the same time that the arc completes. For example, if you set W, X, Y, Z, A, B targets (in that order), and then do the 'A' command, then the A and B axes will be the ones that have the arc draw on them (with the real 'A' axis being treated as a virtual 'X' axis from the point of view of the arc, and the real 'B' axis being treated as a virtual 'Y' axis from the point of view of the arc).

This command can take a very long time, since it operates by drawing the requested number of straight lines in a "circular" pattern. It sends a status report (formatted identically to that generated by the 'h' command) once the last vector of the arc has been queued. If any new command that would change motion operation is received by the controller while drawing an "arc", then the arc drawing will be stopped at the last vertex that was queued before the new command was received.

The firmware uses an internal table of sines to calculate the correct X,Y locations based on the center location and the current angle. The table provides scale factors accurate to about 1 part in 7,000,000; therefore, the actual coordinates calculated can be off by the greater of (1 part in 7,000,000) or (1 part in the radius). As long as the radius is less than 7,000,000, the values will be correct to within 1 unit of measure; otherwise, they can be somewhat off (they are rounded to the nearest value based on the 1 part in 7,000,000 precision).

Note that execution of the '$A' command will clear the current values saved by the 'd', 'c' and 'S' commands upon completion, and will reset the current X and Y parameters to the last X,Y value requested as part of drawing the arc.

As a simple annotated example (the '*' is sent by the controller as its 'ready for next command' response, the 'G0*' is usually sent when the Arc command has finished queuing its vectors),

```
0X          - Set X and
0Y          - Y center point for the arc and as the arc-based motors
```

```
2000R       - Set target rate of 2000
360d        - Set arc delta for full 360 degrees
1000c       - Tell the system that there will be 1000 line segments to draw
0S          - Begin angle is 0
1000$A      - Radius of Arc is 1000, and draw. This draws a “circle” of diameter 2000
                    steps.
0X          - Reset center back to 0,0
0Y                  (otherwise, new center would be the last point drawn)
0d
4c          - This time, just set 4 lines in "arc"
0S          - Again, start at 0 degrees
360d        - Again select a full circle
3000$A      - Draw the 3000 unit radius arc;
                    since done in 4 steps, it is really a square!
```

Note also that the ‘A’rc command can be used to draw a line of a given length (within the limits of rounding and the 1:7,000,000 restriction, above) at any angle from the current location. This can be done by specifying the begin angle as the desired value, the ‘c’ount as 0, and the center X and Y values as the current location. For example,

```
250X        - Set X and
300Y        - Y center point
0c          - Tell the system that there will be 0 steps to draw;
                    we only go to the ‘start’ location
45S         - Begin angle is 45 degrees
945$A       - Radius of Arc is 945, and draw. This draws a line of length 945 at a 45
degree angle
```

This allows you to easily move your motors to the real “start” position of an arc, by first doing a matching arc sequence with the count being 0. This greatly simplifies design of pen-plotter-like systems.

As an optimization, the $A command also supports comma-and-semicolon based setting of its parameters (except for the X, Y center point). When used, a single ';' argument may be specified before specifying the comma-based values, then the $A. Only one '*' response is sent back for these commands, so you have far less handshaking to perform. If an item is omitted (such as not specifying 'R;' or by issuing two commas in a row), then that value will be left as it was last set by a prior '$A' arc command.

The equivalent commands are:

R;S,d,c,a,xA

'R;' - specifies the target rate
'S,' – specifies the start angle
'd,' – specifies the delta angle
'c,' – specifies the count of vectors or the vector length
'$a' – specifies the end radius
'x$A' – specifies the start radius and draws the arc

For example,

```
0X
0Y
2000;0,360,-20,1000$A
```

would request drawing an arc using the X and Y motors centered at X,Y = (0,0), with a rate of 2000, start angle of 0, delta angle of 360, vector length for each arc segment of 20, of radius 1000. Since the $a parameter was not specified, the secondary radius would match the requested radius UNLESS a prior '$a' command had been given and no intervening '$A' command had been issued.

The commands that use the arc function in the scripting system are DrawArc, DrawArcWithRate, DrawSpiral and DrawSpiralWithRate.

$a – Set arc end radius, for spiral generation (DrawSpiral)

The '$a' command is used to specify the ending radius to attain while generating the arc sequence. Note that the code uses the smaller of the two radii ($a or $A value) when it is instructed to calculate the delta arc angle through use of a vector length calculation (a 'c' value that is negative). For example,

```
0X      - Set X and
0Y      - Y center point for the arc and as the arc-based motors
2000R   - Set target rate of 2000
360d    - Set arc delta for full 360 degrees
1000c   - Tell the system that there will be 1000 line segments to draw
0S      - Begin angle is 0
2000$a  - Ending radius is 2000
1000$A  - Radius of Arc is 1000, and draw.
```

This draws a spiral starting with a diameter of 2000, and ending with a diameter of 4000.

Please note that the angles between each vector are constant in all arc motions, be they true arcs or spirals. This means that the vector lengths on spirals will change in direct ratio to the radius ratio (i.e., the vector length for one line segment at radius 1000 will be 1/2 of the vector length at radius 2000).

The scripting system functions that use this command are DrawSpiral and DrawSpiralWithRate.

A – Set next A ordinate (SetRequestedALocation)

This command sets the next A value to use on a 'G', 'g', '$A' or '='command.

For example,

100A

Would cause the next A parameter value to be 100.

The scripting system function that uses this command is SetRequestedALocation.

a – Read A/D channel (ReportAToDValue, ReportAToDFloatValue)

This command allows you to read one of the three available 12 bit A/D input channels, as requested by the parameter (x). The given input must already be configured as an analog input through use of the ‘t’ command in order for this function to not report a 0 result. The voltage range is 0 to 3.3 volts, with 3.3 volts mapping into a raw reading of 4095.

The firmware takes 16 samples on the requested channel, and reports the average of those samples.

```
0a – RDY, when configured as an analog input
1a – NXT, when configured as an analog input
2a – IO2, when configured as an analog input
3a – <reserved>, always reports 0
4a – Chip temperature, as a raw reading
5a – Chip temperature in Celcius * 10
6a – Chip temperature in Fahrenheit * 10
7a – Chip temperature in Kelvin * 10
+8 – Report value as floating point.
       For RDY, NXT and IO2, this is a 0 to 3.3 volt value.
       For temperature, it is simply the temperature,
              in the appropriate units.
```

The report consists of data in the following form:

a,#CH,#AD

where:

'a' is the report type; in this case, 'analog data report'

'#CH' is the channel being reported

'#AD' is the actual reading for the channel

For example,

```
0a
```

Could report

```
a,0,2820
```

which would mean 'the signal on RDY had a raw A/D value of 2820.

while

```
8a
```

would report

```
a,8,2.271906
```

on the same reading (which means that the RDY signal is at 2.27 volts).

A reading of 4095 maps into 3.3 volts.

The scripting system function that uses this command is ReportAToDValue. The StepperBoard Class Library system adds the function "ReportAToDFloatValue" as an additional simplification to for reporting in the floating point mode (i.e., it automatically adds the '8' to the channel request).

B – Set next B ordinate (SetRequestedBLocation)

This command sets the next B value to use on a 'G', 'g', '$A' or '='command.

For example,

```
100B
```

Would cause the next B parameter value to be 100.

The scripting system function that uses this command is SetRequestedBLocation.

b – Select the communications baud rate (SetBoardBaudRate)

The '**b**' command is used to change the communications baud rate from its current value to a requested new value. The factory default setting is 9600 baud, or

9600b

The code takes the rate that you request, and calculates the closest rate to that which it is capable of supporting. It then echoes that rate back, in a response formatted as:

b,#

where '#' is the real baud rate that the code will be able to use. After it completes sending of the final '*' in its response, it will then reset itself to operate at the newly specified rate.

For example, to set the firmware to communicate at 19,200 baud, you would send:

19200b

and you would get the response:

b,19200

*

After you receive the '*', you would then reset your serial system to operate at 19,200 baud.

Suggested values for the baud (which are used exactly) are:

```
2400
4800
9600
19200
38400
57600
115200
```

Many other rates are supported; however, the above list contains the most commonly used values.

Please be forewarned! If you use a non-standard value, or a value above 19,200, then our Stepperboard class libraries (both the Active-X and the Visual Studio .NET systems) may not be able to 'find' the board using an automatic scan. You will need to directly tell the class library what baud rate and what port to use for your application.

If you attempt to set the baud rate to an unsupported value, and ignore the response which contains the real value used, you may not be able to talk to the board. Simply power it off and back on: it will revert to the baud rate that was active the last time that you performed the special '-123456789e' save state command, or to the 9600 baud factory default if you have never saved the firmware state. If you have saved a non-standard baud rate using the save-state command and you forget what you used as the baud rate, you will have to perform the factory-reset action, as described elsewhere in this manual, in order to revert the firmware to a known standard state.

The scripting system function that executes this command is SetBoardBaudRate.

C – Set TTL lines high that are set in this command (QueueSetIOPortsHigh, SetIOPortsHigh)

'C' is used to set the requested TTL signals high which are currently enabled. It is bit encoded as:

Bit	***Value***	***Signal***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	BO1
28	+268435456	BO2

Add the decimal value for each bit that you want to set to '1' (high), and issue the command on the sum. Those bits will get set: the others will be unchanged.

Note that you need to use the 'T' command (T – limit, slew switch control and motor driver enable, SetLimitSwitchEnables) in order to enable access to slew and xO1/xO2 lines, and the 't' command in order to enable access to the RDY, NXT, SI2, SO2 and IO2 lines.

For example, to set the IO2 line high, you could issue the command:

```
16C
```

The remaining TTL lines would be unaffected.

If the value requested is negative (i.e., "-32768D" in the above example), then the request is inserted into the motion queue. It is then executed when that element comes up in the sequence processor, thus allowing a precise setting of a TTL signal when a given vector has completed.

There are several other commands which may be used to monitor and control the output port levels:

- C – Set the TTL lines high that are '1' in 'x'
- D – Set TTL lines to 'x'

- T – Set up access to the slew and xO1/xO1 lines
- U – Set the TTL lines low that are ‘1’ in ‘x’
- u – Set the I/O directions for the slew lines
- The current input values for the limit and slew switches can be retrieved with the '9?' command, as described in the Other report values section.

The scripting functions that implement this command are QueueSetIOPortsHigh and SetIOPortsHigh.

c – Define the arc Count of line segments (DrawArc and DrawSpiral)

‘c’ is used to define the number of line segments or the length of each line segment to draw as part of the ‘Arc’ command. A value of 0 causes the system to go just to the start point defined by the begin angle, the center X,Y, and the arc radius.

Positive values are used to specify the count of line segments.

Negative values are used to specify the approximate length of each line segment of the arc. The code calculates the delta angle to use based on the arc length approximating the requested value.

After completion of the ‘A’rc command, the current ‘c’ value is set to 0.

The commands that use the arc function in the scripting system are DrawArc, DrawArcWithRate, DrawSpiral and DrawSpiralWithRate.

D – Set TTL Lines to the selected values (QueueSetIOPorts, SetIOPorts)

'D' is used to the TTL ports that are currently enabled as TTL outputs to the requested values. It is bit encoded as:

Bit	***Value***	***Signal***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	BO1
28	+268435456	BO2

Add the decimal value for each bit that you want to set to '1' (high), and issue the command on the sum. Those bits will get set: the others will be set to low.

Note that you need to use the 'T' command (T – limit, slew switch control and motor driver enable) in order to enable access to slew and xO1/xO2 lines, and the 't' command in order to enable access to the RDY, NXT, SI2, SO2 and IO2 lines.

For example, to set the IO2 line high and all other lines low, you could issue the command:

```
16D
```

(assuming that IO2 was configured as a TTL output).

If the value requested is negative (i.e., "-32768C" in the above example), then the request is inserted into the motion queue. It is then executed when that element comes up in the sequence processor, thus allowing a precise setting of a TTL signal when a given vector has completed.

There are several other commands which may be used to monitor and control the output port levels:

- C – Set the TTL lines high that are '1' in 'x'
- D – Set TTL lines to 'x'
- T – Set up access to the slew and xO1/xO1 lines

- U – Set the TTL lines low that are '1' in 'x'
- u – Set the I/O directions for the slew lines
- The current input values for the limit and slew switches can be retrieved with the '9?' command, as described in the Other report values section.

The script functions that use this command are QueueSetIOPorts and SetIOPorts.

d – Define the arc Delta angle (DrawArc and DrawSpiral)

'd' is used to define direction of travel for the arc, and the total arc angle relative to the start angle. This is the signed amount (in degrees) to add to the angle set by the 'S' command to reach the end point for the arc. The sign defines the direction of drawing: positive draws counter-clock wise (increasing angle), negative draws clockwise (decreasing angle).

It is legal to specify angles greater than 360; this will simply cause the arc to continue on, which is quite useful when 'drawing' a spiral (see the '$a' parameter).

The commands that use the arc function in the scripting system are DrawArc, DrawArcWithRate, DrawSpiral and DrawSpiralWithRate.

E – Report queue element <x> (SetReportQueueElementSnapshot)

This is used to request a comma-separated listing of the data for the selected queue element.

Values that are >= 0 (currently, values of 0 to 99) directly connect to the fixed array of queue values; this can be used to manually read any element of the queue (usually for diagnostic purposes).

There are four reserved values which report special 'queue' elements:

-1: Most recent element that has been (or is being) processed

-2: Prior GOTO element that was last queued

-3: Current data on GOTO element currently being built for queue

-4: Current location data for all motors, as an instant snapshot

The data is reported as a coma separated list, of the form

E,element,seq,cmd,fparam,lp0,lp1,lp2,lp3,lp4,lp5,lp6

where

element: The requested element number (such as '-1')
seq: a sequential number associated with each queued command
  Note: on the -4E query, this will be the NEXT sequential
    number that will be used on the next queue element.
cmd: The internal command ID: '1' is a vector (goto) command
  For real queue elements (>= 0),
  if the cmd value is positive, it has not yet been processed.
  If it is negative, then it has been pulled from the queue.
fparam: the floating point value associated with this element
lp0 to lp6: 32 bit integer values associated with this element

The single element type that is of most interest is the one with the 'cmd' value being 1. This is a vector command, and assigns the remaining parameters as:

fparam: target rate ('R' command)
lp0: vector length (parameter to 'G')
lp1: W motor target
lp2: X motor target
lp3: Y motor target
lp4: Z motor target
lp5: A motor target
lp6: B motor target

Note: Under the scripting system, the above functionality is found in the following calls:

SetReportQueueElementSnapshot (extracts the above data into an internal array)
ReportQueueElementID
ReportQueueElementCurrentExecutionID
ReportQueueElementCommand
ReportQueueElementFloatParameter
ReportQueueElementLongParameter

e – Write EEProm data (EEPromReset, EEPromSaveCurrentSettings, EEPromSaveUserSettings, EEPromWriteData)

The ‘e’ command is used to rewrite the EEProm data on the board. The 'e' command is ignored if you fail to provide a parameter (i.e., 'e' alone results in the board ignoring the command, while '0e' causes the normal 'e' operation with a parameter of '0').

The data value passed is used to control what kind of write action is to be performed, and is range-based. The data may be written and erased at least 100,000 times.

Value Range	Use
>=0	Write indicated value into the current user-data ‘eeprom’ write address, then increment that address for the following write. This allows access to 32 words (128 bytes) of user-programmable memory
-0 to -31	Specify next write address for user data write (above) NOTE: '-0' normally has to sent by your code as a special action – if you use an integer to hold the value and send the integer, it is likely that your language will treat '-0' as the same as '0', and thus will send it as a 0 without the leading minus.
-123456788	Write user portion of EEProm with the buffered user EEProm data (the 32 user elements, plus the user startup command). If the EEProm is currently empty or invalid, this also writes the control portion (so it acts identically to the “-123456789e” command if the current EEProm contents are invalid).
-123456789	Write control and user portion of EEProm with current state of system. This saves all savable parameters (such as ramp rate, run rate, microstep size, I/O configuration, baud rates and so on) for use on the next reset of the board. It is also the only way to actually transfer the user paramaters into the EEProm
-987654321	Erase the complete EEProm, restoring the system to factory defaults on the next boot.
Other	Illegal – firmware generates a verbose error response

Writing User data

The firmware supports 32 32-bit words (128 bytes) of user programmable EEProm. After a reset, the next EEProm write address is set to a copy of the first word of user data (User data address 0); from then on, each successive ‘e’ command with a data value of 0 to 4294967295 will write to the current user data address, and will then increment that address for the next write. If the new address would exceed 31, it stays at location 31.

To read that data back, use the ‘?’ command with a value of 16384+user.address. The code will reset the user-data access location to the desired address and report the current contents. It will also leave the pointer at that location, so that the next user data write will overwrite the location.

Alternatively, you may specify the next write address by using an ‘e’ command with a data value of -user.address. This simply resets the current address to the desired value. **NOTE: '-0' normally has to sent by your code as a special action – if you use an integer to hold the value and send the integer, it is likely that your language will treat '-0' as the same as '0', and thus will send it as a 0 without the leading minus**.

For example: to read and then update user data location 3, you could issue the commands:

```
16387?
     The board could respond with “X,16387,10” and an ‘*’
```

```
25e
     The board would respond with "*"
16387?
     The board would respond with “X,16387,25”
```

To simply write a ‘5423’ to location 15, you could issue the commands:

```
-15e
     The board would respond with just an ‘*’
5423e
     The board would respond with ‘*’
```

Note that the code is intelligent enough to not actually erase and rewrite the data if the new value matches the old value.

The data is not truly transferred into the User EEProm storage until the either of special "memorize" commands are issued, "-123456788e" or "-123456789e". This is to reduce wear on the storage medium. Normally, you would set all of your user parameters, and then issue the appropriate memorize command.

Saving user EEProm and startup data

The current user data (the 32 array elements plus the user startup command) may be saved by issuing the command:

-123456788e

If the current EEProm data is invalid (i.e., has never been saved, has been corrupted, or is invalid due to a firmware update), then this command will actually operate as if the “-123456789e” command has been requested, and will thus save everything that is capable of being saved.

Saving the current firmware settings

The current firmware settings may all be saved by issuing the command:

-123456789e

This saves everything that is capable of being saved: this includes such items as the current USER values (the 32 parameters accessed via the 'e' command), ramp rates, start/stop rates, A/D control values, and all other key I/O port information. Items that are not saved are transient data such as current I/O port contents and current motor locations.

The new settings are used the next time the board is powered on, and the next time that a board reset (either soft via the ‘!’ command or hard via the RST input) is issued.

Firmware settings that are saved by “-123456789e”

The settings that are saved by the save firmware settings command are:

- b – set baud rate
- C, D and U – User programmable TTL output values
- F – Backlash adjustment
- H – Current during motor motion
- j – Microstep size for each motor
- J – Dac voltages
- $K – Asynch Stop OK rate
- $P – Asynch Slope rate
- $R – Asynch Run rate
- $r – Potentiometer rate scale factors for asynch mode
- T – Limit and slew mask bits and motor asynch/vector modes
- t – configure I/O port direction and use
- V – verbose communications control, hex mode report control
- N – motor current when motor idle

The scripting system commands that use this function are EEPromReset, EEPromSaveCurrentSettings and EEPromWriteData.

Restoring firmware settings to Factory Default values

The firmware EEProm may be reset to factory defaults by two methods.

1. Issue the '-987654321e' command.
2. Do an "EEPROM RESET" hardware strap startup as is described below.

Both methods erase the EEProm flash memory, including any data that you have saved using the above "user data" write commands. This sets all of the memory to have a value of 0.

The first method purely erases the memory. It does not otherwise affect the current state of the board – that is to say, your current baud rates, start/stop rates and so on will remain unchanged until you reset the board (at which point the factory default settings become active).

The second method ("EEPROM RESET") is your emergency recovery method to allow you to regain control of a board which has unknown settings and a saved baud rate that cannot be discovered using our standard tools. The technique is:

1. Power off the board
2. Disconnect everything from the board except for the power connection
3. Use a standard 0.1" shunt to jumper the "SI" and "SO" pins together that are on the connector labelled "IO" (the one that also has the NXT and RDY signals).
4. Power the board on
5. Wait 3 seconds
6. Power the board off
7. Remove the shunt from SI and SO
8. Reconnect your motors and wires as needed

The board should now be restored to its factory-default settings. If you still cannot 'talk' to the board, please contact us for further instructions.

F – Specify the motor backlash amount (SetMotorBacklash)

'F' is used to specify (in microsteps) the motor backlash amount. The currently selected motor(s) (see the 'm' command) all get their backlash set to the specified value. This value may be from 0 to 65535.

Whenever a motor's direction of spin changes, the motor's position gets preadjusted by the backlash amount so that the system will correctly wind-up (tighten) the backlash. This action is fully transparent to external code: it occurs at any time that a new spin direction is requested for each motor.

Note that the code does NOT insert a separate 'windup' vector to take up the backlash; rather, it increases the number of steps by the backlash amount whenever the motor's spin direction changes.

The scripting system command for this action is SetMotorBacklash.

f – Specify rate scale factor (SetRateScaleFactor)

'f' is used to dynamically specify a fixed point multiplier to apply to the current target rate.

This factor may be applied at any time, including when motion is under way. The code will automatically ramp the current rate up and down accordingly in order to attain the requested new rate, through use of the 'ramp rate' parameter.

For example, if your current rate ("R") is set to 2000, issuing the command

```
0.5f
```

will tell the system that the real rate is 0.5 * R, or 1000. If the motors are already moving faster than 1000, they will slow down to the 1000 value.

A special value of '0f' turns off rate scaling -- this is identical in results to a value of '1f'.

The scripting system command for this action is SetRateScaleFactor.

G – Queue the request to Go to the currently requested absolute vector position (GoToCurrentRequestedLocation, GoToCurrentRequestedLocationWithVectorLength, GoToLocation, GoToLocationWithVectorLength, GoToLocationWithVectorLengthAndRate)

This command will queue the new vector location (from the W, X, Y, Z, A and B commands) as the next location to target. The vector location is treated as an absolute motion (use 'g' to queue a relative motion request). Note that the vector motion only affects motors that are enabled for vector motion: any motors that are configured (via the 'T' command) as asynchronous motors will not respond to this command.

If the current motor motion mode is trapezoidal (the default), then the new X,Y location will be queued as a continuation of the current "Multi-vector chained motion" (see the "M" command). If there is no current multi-vector chained motion active, then the new vector will consist of the complete ramp up, run at target rate and ramp down then stop action for trapezoidal motion processing.

If the current motor motion is non-trapezoidal (via the 'rate' command being negative), then the new request will simply be queued as needed.

The optional parameter to the 'G' command specifies the nominal vector length to use for all rate-and-distance calculations. The parameter defines a pseudo axis which receives the actual motion request (in terms of applying the rate and distance calculations), with the other axes being scaled to match. If the parameter is not present it will be reset to the largest of the distance being traveled on the six axes.

The software will:

- Calculate the direction and distance of travel for all motors, and the correct relative rates for the actions
- Queue the request to be started upon completion of the current motion
- Send back a count of elements available in the queue
- Send back the "*" acknowledgement character

For example,

```
1000X
-25687Y
30000G
```

Would:

1. Set the next X value to 1000
2. Set the next Y value to –25687
3. Set the nominal target vector length to 30,000 for rate calculations
4. Queue a GOTO on motor location 1000, -25687
5. Note that the firmware would respond with the queue available count (in this case, followed by the '*' acknowledgement.

Note that the code will send back the queue count and "*" acknowledgement character as soon as the request has been queued; ***the code will wait until a queue slot becomes available before it sends the queue count and '*' response***. If it receives another character while waiting for the slot, then the new GoTo action is aborted (i.e., the new location is never queued).

Communications performance enhancement – comma and semicolon arguments

As an optimization, the G and g commands also support comma-and-semicolon based setting of their parameters. When used, a single ';' argument may be specified before specifying the comma-based values, then the G or g. Only one '*' response is sent back for these commands, so you have far less handshaking to perform. If an item is omitted (such as not specifying 'R;' or by issuing two commas in a row), then that value will be left as it was last set by a prior related command. The final parameter (after the last comma) always specifies the length of the pseudo-vector: set a value of 0 if you want the code to use the maximum motion length as the vector length.

The equivalent commands are:

```
R;W,X,Y,Z,A,B,xG
```

'R;' - specifies the target rate
'W' – specifies the W motor target
'X' – specifies the X motor target
'Y' – specifies the Y motor target
'Z' – specifies the Z motor target
'A' – specifies the A motor target
'B' – specifies the B motor target
'xG' – specifies the vector length and queues the vector

For example,

```
2000;10,20,30,40,50,60,100G
```

would request generation of a vector of length 100 at rate 2000 going to location <10,20,30,40,50,60>.

The scripting system commands that support this functionality are:

- SetAddressMode (to define whether motion is relative or absolute)
- GoToCurrentRequestedLocation
- GoToCurrentRequestedLocationWithVectorLength
- GoToLocation
- GoToLocationWithVectorLength and
- GoToLocationWithVectorLengthAndRate.

g – Queue the request to Go to the currently requested relative vector position (GoToCurrentRequestedLocation, GoToCurrentRequestedLocationWithVectorLength, GoToLocation, GoToLocationWithVectorLength, GoToLocationWithVectorLengthAndRate)

'g' is fully identical to 'G', except that it treats the requested vector location <W,X,Y,Z,A,B> as relative to the current location. See 'G' for details about the command.

The scripting system commands that support this functionality are:

- SetAddressMode (to define whether motion is relative or absolute)
- GoToCurrentRequestedLocation
- GoToCurrentRequestedLocationWithVectorLength
- GoToLocation
- GoToLocationWithVectorLength and
- GoToLocationWithVectorLengthAndRate.

$G – Do a GoTo action on the selected asynchronous motors (AsynchGoToLocation)

$G allows you to specify motion on any motor that is currently selected via the 'm' command that is also currently defined as an asynchronous motor via the 'T' command. The parameter is the exact target location for the requested goto action.

Related asynchronous commands are:

$K – Set asynchronous motor start/stop rate
$P – Set asynchronous motor ramp rate
$R – Set asynchronous run rate

For example, to make motor X go to location 1000 at a rate of 300, you could issue the commands:

```
2m
300$R
1000$G
```

This motion is fully independent of any queued vector motion.

The scripting system command that implements this function is AsynchGoToLocation.

H – Specify Motor Current When Moving (SetCurrentWhenMotorInMotion, SetO1O2WhenMotorInMotion)

“**H**” is used to specify the motor current to use when motor motion is under way. A separate current may be specified for each motor.

On the BC6D20/BC6D25, the units used for the “H” command are milliamps normally milliamps; thus, a command of ‘500H’ is requesting that the windings be operated at 0.5 amps when the motors are in motion. If H is set to 0, this fully disables the motor as no current will flow (the motor is “free”).

On the SD6DX, this controls the automatic setting of the xO1 and xO2 output lines (x being the motor label). Bit 0 is for xO1, while bit 1 is for xO2.

A separate setting may be specified for each motor. By default, at power-on, the “H” value for the BC6D20/BC6D25 is set to **500**, which selects 0.5 amps per winding. On the SD6DX, the default setting is **1**, which has O1 high and O2 low.

BC6D20/BC6D25 notes:

Values for H that are below 100 mA are often not very accurate; this is what specifies the minimum current that a given version of our board can generate.

The actual current defined by the 'H' command is dependent on the current scale factor as defined by the 'o' command. Please see the 'o' command for details of changing the scale units for the H command. By default, the normal units for 'H' are milliamps, so '500' maps into 0.5 amps.

Additionally, as of firmware version 5.29, if a decimal point is detected in the parameter for the ‘N’ function, the units are treated as in amps, not milliamps. Therefore, as of version 5.29, the following commands are equivalent:

```
500H
0.5H
```

Note that the actual current delivered is probably no more accurate than 10-20%.

On the BC6D20 and BC6D25, the scripting system command for this is SetCurrentWhenMotorInMotion.

SD6DX notes:

On the SD6DX, the “T” command (SetLimitSwitchEnables) may be used to override the automatic control of the O1 and/or O2 lines for each motor. If you desire, you may disable automatic setting of the O1/O2 values, and program the values manually with the standard TTL output commands.

The scripting system command that implements this function on the SD6DX product is SetO1O2WhenMotorInMotion.

h – instant queue and motion status report (IdleWait, IdleWaitForFullyStopped)

‘h’ always requests an instant report on the current queue and motor status. It always returns a multi-character status response (the same one generated by I, i, G, g and A), as:

s#*

where:

s is the status:

‘A’ – The board is currently waiting on execution of an “arc” (multi-line segment) request

'D' – currently executing a 'delayed pause' action ('p' with a negative parameter)

‘G’ – The board is currently moving to a new goto target

‘I’ – motion is complete, the board is idle

'i' – vector motion is complete, but asynchronous motion is still occurring.

‘P’ – The board is ‘paused’ (due to limit switch actuation or the ‘q’ command); no further motion will occur until ‘r’ or ‘Q’ is issued

'S' – board is stopped from a TTL stop event (see the 't' command); no further motion will occur until the 'Q' command is issued

# is the count of elements left in the motion queue. In the current firmware, this number varies from 0 (queue is full, no new queue commands can complete) to 100 (queue is empty). If a value of ‘-1’ is reported, the last queued request was dropped due to its having been issued when the queue was already full.

* reports completion of the status report.

The ‘h’ command may be given at any time (including when waiting on a blocked command, such as an ‘X’ when the queue is already full). It will report the current status, and then the command will continue its execution. Note that if you then want to have a new prompt (an ‘*’) generated when the pending command completes enough for a new command, you would have to follow the ‘h’ command with an ‘i’ command, in order to get the delayed completion report.

The “h” command gives you a method of safely resynchronizing with the board – it immediately sends back the complete status report, after which you can decide if you need to send an ‘i’ or ‘I’ for a delayed report, or if the controller is ready for new commands.

The scripting system commands that use this function are IdleWait and IdleWaitForFullyStopped.

I Wait for motor 'Idle'

This allows your code to 'wait' for the motors to be fully idle. It also provides you with the reason as to why the motors are idle. (See the 'h' command for a variant which generates the same response with no wait). Sending an 'I' is always safe (even if you have not received an '*' from another command); this allows you to easily 'poll' the board in a multi-port environment. The command also reports the queue count as part of its return, so that you know how many commands may be buffered.

This is normally used to simply wait for either motion to be complete, or to detect that the motors have stopped due to a limit switch action (they are paused).

The report is sent back once the motors are idle (or paused); it is formatted identically to that of the 'h' instant status report (above).

In the scripting system, this functionality is implemented via the IdleWaitForFullyStopped command.

i – Wait for command queue space

This allows your code to 'wait' for queue space to be available for new motion commands (such as 'G' and '$A'). It also reports what main type of action is occurring; this allows your code to confirm whether a new command can be sent. Sending an 'I' or 'i' is always safe (even if you have not received an '*' from another command); this allows you to easily 'poll' the board in a multi-port environment. The command also reports the queue count as part of its return, so that you know how many commands may be buffered.

'i' waits until there is at least one queue element available (or the motors are paused) before it sends back its report, at which point it sends the same report as that generated by the 'h' command.

j – set Microstep size for this pair of motors (SetMicrostepSize) – BC6D20/BC6D25 only

On the BC6D20/BC6D25, the 'j' command allows the microstep size to be separately specified for each pair of motors (as selected by the 'm' command). As with the reset command, the microstep size must be one of the perfect powers of 2 from 1 to 16 or 32 (1, 2, 4, 8, 16, 32), and is expressed in units of 1/16th of a step on the BC6D20 and in units of 1/32nd of a step on the BC6D25.

Due to hardware limitations (i.e, we ran out of signals), the microstep size is assigned to each odd/even pair of motors. That is to say, motor pairs (W,X), (Y,Z) and (A,B) always have matching microstep sizes. If a differing microstep size is requested for two motors in a pair, the last one specified is used on both motors.

WARNING! Specification of the microstep size is a 'major event' for the board. You will get incorrect operation if you issue this command while any kind of motion is under way. Also, any non-0 locations will no longer have much meaning, since their units will have changed.

On the BC6D20, 1/8th step motion has a limit of 250,000 microsteps/second, while all other step sizes support 500,000 microsteps/second.

On the BC6D25, the maximum step rate limit is always 250,000 microsteps/second.

The scripting system command for this function is SetMicrostepSize.

Note: The SD6DX firmware ignores the lower-case 'j' command.

J – set DAC voltages (SetDacInMillivolts and SetDacInVolts) – SD6DX only

On the SD6DX, the 'J' command allows control of the output voltages on the dual DAC outputs.

The command is comma separated based: the values terminated by commas specify which dac to set (and whether to memorize the dac setting), while the value just before the J specifies the voltage.

The firmware looks for a period (".") in the value sent with the 'J' command. If there is no period, then the units are millivolts. If there is a period, then the units are volts.

For example,

```
0,3500J
```

would set V0 to 3.5 volts (3500 millivolts), while

```
1,1.75J
```

would set V1 to 1.75 volts (1750 millivolts).

The scripting system commands for this function are SetDacInMillivolts and SetDacInVolts.

The BC6D20 and BC6D25 ignore the upper case "J" command.

$J – Set external ISODAC voltages (SetExtDac, SetExtDacInMillivolts and SetExtDacInVolts)

On all of the six axis products, an external board called the "ISODAC" may be connected via the optional 10 pin header in the middle of the board. This board provides for a 0-5 or 0-10 volt isolated DAC output, suitable for use in such items as spindle motor control.

As with the "J" command, the firmware looks for a period (".") in the value sent with the 'J' command. If there is no period, then the units are millivolts. If there is a period, then the units are volts.

The command is comma separated based: the values terminated by commas specify which dac to set (and whether to memorize the dac setting), while the value just before the $J specifies the voltage.

The mapping of the dac address value is:

Address	**ISODAC Board #**	**Use**
0	0	Just set the dac
1	0	set dac and memorize value
2	1	Just set the dac
3	1	set dac and memorize value
4	2	Just set the dac
5	2	set dac and memorize value
6	3	Just set the dac
7	3	set dac and memorize value

For example,

```
0,3,2000$J
```

would set the dacs on boards 0 and 1 to 2 volts, and would memorize that setting on board 1.

NOTE: Please see the 't' command for information on how to configure the scale used in this function (whether the board is set up as 0-5 or 0-10 volt output).

The scripting system commands for this function are SetExtDac, SetExtDacInMillivolts and SetExtDacInVolts.

$K –Set the asynchronous motor "Stop oK" rate (AsynchSetStopOKRate)

This defines the motor rate at which the selected motors are considered to be "stopped" for the purposes of stopping or reversing directions. It defaults to the default of '80' if a value of 0 is given.

By default, this is preset to "80" upon startup of the system. This means that, whenever a stop is requested, the motor will be treated as "stopped" when its stepping rate is <= 80 microsteps (5 full steps) per second.

For example,

```
100$K
```

sets the stop rates for the currently selected motor(s) to be 100 microsteps per second. Any time the current rate is less than or equal to 100, the motor will have the ability to stop instantly.

To set the rate such that the motors always immediately start and stop at the desired rate ('R') setting, issue the command:

```
250000$K
```

This sets the start/stop rate to the maximum rate supported by the firmware, thus blocking any automatic ramp operations.

The scripting system command for this function is AsynchSetStopOKRate.

K –Set the vector "Stop oK" rate (SetStopOKRate)

This defines the vector rate at which the motors are considered to be "stopped" for the purposes of stopping or reversing directions. It defaults to the default of '80' if a value of 0 is given.

By default, this is preset to "80" upon startup of the system. This means that, whenever a stop is requested, the motor will be treated as "stopped" when its stepping rate is <= 80 microsteps (5 full steps) per second.

For example,

```
100K
```

sets the stop rates for the currently selected motor(s) to be 100 microsteps per second. Any time the current rate is less than or equal to 100, the motor will have the ability to stop instantly.

To set the rate such that the motors always immediately start and stop at the desired rate ('R') setting, issue the command:

```
30720K
```

This sets the 'Stop oK' rate to the maximum possible step rate, and thus will prevent all ramping behaviors of the code.

The scripting system command for this function is SetStopOKRate.

k –Set the current instantaneous motor rate (QueueRateSet)

If the current rate mode (see the 'R' command) is non-trapezoidal (i.e., a negative rate has been requested), then the 'k' command instantly sets the current rate to the requested value, with no intervening rate ramping. This mode can be used to force start conditions when in the non-trapezoidal mode of motion control.

The scripting system command for this function is QueueRateSet.

L – Latch Report: Report current latches, reset latches to 0 (ReportLatchedData)

The "L"atch report allows capture of key short-term states, which may affect external program logic. It reports the "latched" values of system events, using a binary-encoded method. Once it has reported a given "event", it resets the latch for that event to 0, so that a new "L" command will only report new events since the last "L".

The latched events reported are as follows:

Bit	***Value***	***Description***
0	+1	W- limit reached during a W- step action
1	+2	W+ limit reached during a W+ step action
2	+4	X- limit reached during a X- step action
3	+8	X+ limit reached during a X+ step action
4	+16	Y- limit reached during a Y- step action
5	+32	Y+ limit reached during a Y+ step action
6	+64	Z- limit reached during a Z- step action
7	+128	Z+ limit reached during a Z+ step action
8	+256	A- limit reached during an A- step action
9	+512	A+ limit reached during an A+ step action
10	+1024	B- limit reached during a B- step action
11	+2048	B+ limit reached during a B+ step action
12-15		Reserved for two more motors
16	+65536	System power-on or reset ("!") has occurred
17	+131072	Motion is blocked via TTL input (NXT input is at its 'block motion' level, if so configured)
18	+262144	If set, the precision 50 PPM clock is being used as the clock source. If clear, the internal 3% oscillator is being used as the clock source.
19-23		Reserved: currently all 0
24	+16777216	BC6D25: W motor driver is reporting a fault
25	+33554432	BC6D25: X motor driver is reporting a fault
26	+67108864	BC6D25: Y motor driver is reporting a fault
27	+134217728	BC6D25: Z motor driver is reporting a fault
28	+268435456	BC6D25: A motor driver is reporting a fault
29	+536870912	BC6D25: B motor driver is reporting a fault

For example, after initial power on,

```
L
```

Would report

```
L,65536
*
```

If you were then to do an X seek in the "-" direction, and you hit an X limit, then the next "L" command could report:

```
L,4
*
```

The scripting system function for this is ReportLatchedData.

$M – Mark current location or goto marked location (AsynchMarkCurrentLocation, AsynchGoToMarkedLocation)

For asynchronous motors, the $M command allows you to mark the current location or to goto a previously marked loction.

0M	Mark the current location
1M	Go to the previously marked location

The scripting system commands for this action are AsynchMarkCurrentLocation, and AsynchGoToMarkedLocation.

M – Start Multi-vector chained motion (StartMultiVectorSequence)

The M command queues a request for a new chained vector. The parameter passed defines the total length for the vector (the parameters to the following 'G' commands), so that the code can correctly treat the ensuing motion requests as one trapezoidal motion event. The firmware automatically adjusts this length for any motor backlash encountered as the motion progresses. The motion is considered to be complete when this total distance has been consumed by the pending vectors (note: the last vector may extend the total distance if the 'M' value is too small).

There are 3 possible sets of values for the M command

> 0: This is the total vector length for the entire set of motion events
= 0: Complete any merged motion here – next 'G' starts a new event
= -1: Start automatic chained vector mode.

The final mode of '-1M' allows for mostly automatic merging of vectors by the code. In this mode, the firmware will keep 'chaining' successive vectors together ('G' and 'g' commands) until a new 'M' command is issued (or a 'Q' is done to abort all motion). If you are not able to 'feed' vectors to the queue rapidly enough to stay ahead of the motion, the code will automatically slow down and speed up the motion as needed to prevent any sudden stops from happening if the queue gets emptied. The '-1M' mode should always be completed with either a '0M' or a '-1M' (starting a new vector), to fully optimize the generated motion.

The >0M mode is far less forgiving. It requires that you keep the queue from emptying by sending data rapidly enough to stay ahead of the motion. If the firmware runs out of motion events while processing a set of vectors when in this mode, the motors will experience an instant stop, followed by an attempt to perform an instant start (at whatever rate had been attained by the time of the instant stop) as soon as another vector makes it into the queue. The effect is likely to be very choppy motion and missed steps! This mode is present for "compatibility" with the NC firmwares on earlier products, and is not recommended for current use.

The scripting system command for this action is StartMultiVectorSequence.

m – Select motors for parameter set and retrieval

This command selects any combination of the motors to be selected as targets for any following '?' commands. The value is bit-encoded as follows:

Bit	***Value***	***Motor Selected***
0	+1	M1: W
1	+2	M2: X
2	+4	M3: Y
3	+8	M4: Z
4	+16	M5: A
5	+32	M6: B

For example,

```
63m
0?
```

Would generate a report about all reportable parameters for all motors.

At power on/reset, all motors are selected for the selected actions.

NOTE: The scripting system does not use the "m" method for selecting motors. Each scripting command that requires a motor selection includes the motor selection as part of the command.

The StepperBoard class library provides this capability using its "SelectMotor" function call.

N – Set motor current for selected motor when idle (SetCurrentWhenMotorIdle, SetO1O2WhenMotorIdle)

The "**N**" command controls whether the currently selected motor(s) has its windings left enabled or disabled once any GoTo action has completed.

On the BC6D20/BC6D25, the units used for the 'N' command are normally milliamps; thus, a command of '100N' is requesting that the windings be left at 0.1 amps when the motors are idle.

On the SD6DX, this controls the automatic setting of the xO1 and xO2 output lines (x being the motor label). Bit 0 is for xO1, while bit 1 is for xO2.

A separate setting may be specified for each motor. By default, at power-on, the "N" value for the BC6D20/BC6D25 is set to **0**, which fully disables the motor as no current will flow (the motor is "free"). On the SD6DX, the default setting is **2**, which has O1 low and O2 high.

BC6D20/BC6D25 notes:

Values for N that are below a board-specific value (such as 100 mA) are not very accurate; this is what specifies the minimum current that a given version of our board can generate.

The actual current defined by the 'N' command is dependent on the current scale factor as defined by the 'o' command; by default, the units are milliamps. This allows you to change the units used for current in order to match special conditions, prior firmwares or prior product offerings.

Additionally, as of firmware version 5.29, if a decimal point is detected in the parameter for the 'N' function, the units are treated as in amps, not milliamps. Therefore, as of version 5.29, the following commands are equivalent:

```
500N
0.5N
```

Note that the actual current delivered is probably no more accurate than 10-20%.

On the BC6D20 and BC6D25, the scripting system command for this is SetCurrentWhenMotorIdle.

SD6DX notes:

On the SD6DX, the "T" command (SetLimitSwitchEnables) may be used to override the automatic control of the O1 and/or O2 lines for each motor. If you desire, you may disable automatic setting of the O1/O2 values, and program the values manually with the standard TTL output commands.

The scripting system command that implements this function on the SD6DX product is SetO1O2WhenMotorIdle.

n – Reset the current queued element counter (SetNextQueueElementLongID)

The 'n' command may be used to reset the queued element counter. This counter is a 32 bit incrementing value, that gets incremented after adding each new element to the queue. Note that it will 'wrap' when it increments from +2147483647: the next value will overflow to -2147483648.

The value that you specify as the parameter to 'n' is the one that will be assigned to the next queued element (for example, the next element generated by a 'G' command).

The scripting system command that implements this function is SetNextQueueElementLongID.

o – Sets scale factor to use for current commands "H" and "N" (BC6D20/BC6D25 only)

On the BC6D20/BC6D25 products, "H" and "N" are used to specify the motor current when motor motion is under way ("H") or complete ("N"). The units used by those commands may be controlled through use of the "o" command.

By default, "o" is set for units of milliamps. This allows the intuitive setting of your motor current in 'obvious' units, such as

```
500H
```

setting the motor current to be 500 mA, or 0.5 Amps.

However, in your application it may be better to use other units. If your code was previously written to operate one of our BC2D15 controllers, you may prefer for the units to match that controller. If you want to run the board in its double-winding full-step mode of operation, you may want to change the units to 'intuitively' describe the per-winding current. Through use of the 'o' command, these goals may be met.

The actual formula for determining the current delivered to the motor is derived as follows:

DAC = the actual dac value generated

SF = scale factor from the 'o' command

I = actual current value targeted by the board

H = current as requested from the H or N command

DAC = H * SF

I = DAC * 2.0/247

so

I = H * SF * 2.0/247

Additionally, several special values have been reserved for compatibility with older boards.

Value	***Description***
0 or 5341	**Units are milliamps**
1	**Units are amps**
2 or 65535	**Units are raw DAC values**
47407	**Units match those used by the BC2D15 series of controllers**

Note that the actual current delivered is probably no more accurate than 10-20%.

$P – asynchronous motor sloPe (number of steps/second that rate may change) (AsynchSetRampRate)

This command defines the maximum rate at which the selected asynchronous motor's speed is increased and decreased. By providing a "slope", the system allows items which are connected to the motor to not be "jerked" suddenly, either on stopping or starting. In some circumstances, the top speed at which the motor will run will be increased by this capability; in all cases, stress will be lower on gear systems and motor assemblies.

The slope can be specified to be from 1 through > 1,600,000,000 microsteps per second per second. If a value of 0 is specified, the code forces it to have a value of 8000.

This value defaults at power-on or reset to 8000 microsteps per second per second. Please note that changing this during a "goto" action will cause the stop at the end of the goto to potentially be too sudden or too slow – it is better to first stop any "goto" in progress, and then change this slope rate.

For example, if we currently have motor X selected, and it is at location 0, then the sequence:

```
250$P

500$R

2000$G
```

would cause the following actual ramp behaviors to occur:

1. The motor would start at its "stop oK" rate, such as 80 microsteps/second
2. It would accelerate to its target rate of 500 microsteps per second, at an acceleration rate of 250 microsteps/second/second.
3. This phase would last for approximately 500/250 or about 2 seconds, and would cover about 500 microsteps of distance.
4. It would then stay at the 500 microstep per second target rate until it was about 500 microsteps from its target location, i.e., at location 1500 (which would take another 2 seconds of time).
5. It would then slow down, again at a rate of 250 microsteps per second, until it reached the stop oK rate. As with the acceleration phase, this would take about 2 seconds.

The total distance traveled would be exactly 2000 microsteps, and the time would be 2+2+2=6 seconds (actually, very slightly less).

The scripting system command that implements this function is AsynchSetRampRate.

P – vector sloPe (number of steps/second that rate may change) (SetRampRate)

This command defines the maximum rate at which the vector motor's speed is increased and decreased. By providing a "slope", the system allows items which are connected to the motor to not be "jerked" suddenly, either on stopping or starting. In some circumstances, the top speed at which the motor will run will be increased by this capability; in all cases, stress will be lower on gear systems and motor assemblies.

The slope can be specified to be from 1 through > 1,600,000,000 microsteps per second per second. If a value of 0 is specified, the code forces it to have a value of 8000.

This value defaults at power-on or reset to 8000 microsteps per second per second. Please note that changing this during a "goto" action will cause the stop at the end of the goto to potentially be too sudden or too slow – it is better to first stop any "goto" in progress, and then change this slope rate.

For example, if we currently have motor X selected, and it is at location 0, then the sequence:

250P

500R

2000G

would cause the following actual ramp behaviors to occur:

1. The motor would start at its "stop oK" rate, such as 80 microsteps/second
2. It would accelerate to its target rate of 500 microsteps per second, at an acceleration rate of 250 microsteps/second/second.
3. This phase would last for approximately 500/250 or about 2 seconds, and would cover about 500 microsteps of distance.
4. It would then stay at the 500 microstep per second target rate until it was about 500 microsteps from its target location, i.e., at location 1500 (which would take another 2 seconds of time).
5. It would then slow down, again at a rate of 250 microsteps per second, until it reached the stop oK rate. As with the acceleration phase, this would take about 2 seconds.
6. The total distance traveled would be exactly 2000 microsteps, and the time would be 2+2+2=6 seconds (actually, very slightly less).

The scripting system command that implements this function is SetRampRate.

p – Pause the controller for the requested time interval (PauseNow, QueuePause)

This command executes a delay for at least x microseconds after receipt, or queues such a request if the 'x' parameter is negative.

1000p

would wait 1 millisecond before sending back its '*' response.

-1000p

would queue a 1 millisecond pause in the current vector sequence; this can be useful if you are also queuing TTL output state changes, in order to control the width of the pulses.

The scripting system commands that implement this function are PauseNow and QueuePause.

Q – Stop all motors immediately, abort queued motions (StopAllMotors, StopVectorMotors, StopASynchMotors)

‘Q’ causes the selected set of motors to be instantly stopped (no ramping is performed), and will abort all pending queued actions if vector motors are to be stopped. **Please note that this action can damage gear systems, since the stop is immediate, with no slow-down behaviors!**

When 'Q' is given with no parameter, the behavior is to instantly stop all motion of all motors, and to purge the vector queue.

If a parameter is given, then the effect of 'Q' depends on the value of the parameter:

0 : Stop both vector motors and asynch motors, purge the vector queue
1: Just stop vector motors, purge the vector queue
2: Just stop asynch motors
3: Same as 0: stop both vector motors and asynch motors, purge the vector queue

The scripting system commands that implement this function are StopAllMotors, StopVectorMotors and StopASynchMotors.

q – Pause motors with optional TTL restart (PauseMotors, PauseMotorsWithTTLRestart, QueuePauseMotorsWithTTLRestart)

‘q’ causes the motors to be ramped to a complete stop, according to the current ramp rate and stepping rate. “Stopped” is defined as “having a step rate which is <= the stop oK rate” (see the ‘K’ command for defining the “stop oK rate”). Once the motors are paused, the system waits for a ‘r’estart (i.e., the Restart command) or a ‘Q’ command (abort all) before it permits any motor motion to occur.

'q' supports an extension that allows an automatic restart via a user-programmed TTL input port. If a parameter is specified on the 'q' command, it is a bit-encoded datum that describes how the 'q' is to behave, as follows:

Bits	**Value**	**Description**
0-2	0-7	Restart action to perform 0-1: No TTL restart ('r' only) 2: Restart when selected input low 3: Restart when selected input high 4: Restart when selected input edge goes low 5: Restart when selected input edge goes high 6-7: Reserved: do not use
3-8	0-9	Bit number from the '9?' query to use as TTL input for source of restart request: the bit numbers are the same as those found in the TTL I/O commands (such as C and D) 0 RDY 1 NXT 2 SI2 3 SO2 4 IO2 5 W- 6 W 7 X- 8 X+ 9 Y- 10 Y+ 11 Z- 12 Z+ 13 A- 14 A+ 15 B- 16 B+

Also, if the value is preceded by a negative sign, then the pause request may be queued. This allows you to queue sequences that will not start until a TTL event (or 'r' request) is issued. Note that a queued pause has one key difference from an immediate pause: unlike the immediate pause (which has a slow-down-and-stop behavior before it enters the paused state), the queued pause is instant. No more motion will occur until the restart is requested. When operating chained vectors, if a queued pause is inserted in the middle of the chain, you will get an instant stop at the point of insertion. **Please note that this action can damage gear systems, since the stop is immediate, with no slow-down behaviors.**

For example, to queue a pause request that will be restarted when the NXT signal is high (level triggered), the command would be calculated with a base value of 3+(8 * 1) → 11 ('3' as the bits 0-2 request for "restart when selected input high", '8*1' as bit 3-8 request (hence scale by 8) for the NXT signal).

```
-11q
```

The code will insert the request into the queue, and will enter the pause state when it executes the queued command. It will then wait for the NXT signal to be high before it continues processing (or for you to issue the 'r' command, or the 'Q' command to abort everything).

To immediately wait for a low-going edge on the IO2 line, the command would be calculated with a base value of 4+(4*8)→36:

```
36q
```

This would cause the firmware to request a pause now (with a slow down action before it actually pauses). It would not start looking for the TTL edge until it was actually paused – an edge that occurs before the motors have stopped will be ignored!

The scripting system commands that implement this function are PauseMotors, PauseMotorsWithTTLRestart and QueuePauseMotorsWithTTLRestart.

$R – Set run Rate target speed for selected asynchronous motor(s) (AsynchSetRunRate)

This defines the run-rate to be used for the currently selected motors. It may be specified to be between 1 and 500,000 microsteps per second (250,000 microstep per second if you are operating at 1/8th step resolution or are using the BC6D25). If a value of 0 is specified, the code switches to use of the potentiometer as its rate source. If a value outside of the limits is specified, then it is accepted, but the code will not operate reliably.

This defines the equivalent number of microsteps/second which are to be used to run the currently selected motor under the GoTo or Slew command. The internal motor position is updated at this rate, using a sampling interval of 500,000 update tests per second.

For example,

```
2m
250$R
4m
1000$R
```

Sets the X motor target stepping rate to 250 microsteps per second, and the Y motor target rate to 1000 microsteps per second.

The power-on/reset default Rate is 800 microsteps/second.

If you are currently executing a targeted GoTo or Slew command which has a specific target location (i.e., "2000$G" or "-300$S"), the new rate will not take effect until the motion has completed. If you are executing a generic "Slew in a given direction" command ("+$S" or "-$S"), the new rate will take effect immediately, and the motor will change its rate to match the request using the current "$P" (ramp-rate) value.

The scripting system command that implements this function is AsynchSetRunRate.

$r – Set rate scaling parameters for asynchronous motor rate control (SetPotRateScales, SetPotRateScalesToDefaults)

The 'Sr' command is used to scale A/D readings into useable rate values when the NXT, RDY or IO2 connections are used as motor rate control inputs (see the 't' command, t – Configure all I/O port directions and use).

By default, if the 'r' set of commands is not given for a given motor, the system is set to scale a '0' A/D reading into a rate of '1', and a 3.3 volt 4095 A/D reading into the maximum rate of 500,000 on the BC6D20 and the SD6DX units, and 250000 on the BC6D25 . Through use of the microcoded 'r' commands, the voltage inputs may be fully scaled into real rates, with both minimum and maximum rates being specified as part of the configuration. Additionally, you may scale a high voltage (such as 3.3 volts) into the minimum desired rate, with a low voltage (like 1 volt) into the maximum rate that you want.

The 'r' command is actually a set of commands, microcoded by their low 3 bits. The 'r' command mostly operates on a set of temporary ("pending") registers, which then become active with the special 'Accept new pending data' command.

Use of 'r' consists of:

1. Reset the 'pending data' to default values, so you know your starting point
2. Issue the appropriate 'r' commands to change from those defaults to new values as needed
3. Issue the 'accept new pending data' command to activate the new scaling
4. To actually use the potentiometer for control, you then need to set the rate of the associated motor to 0 ("0R").

The encoding is as shown in the following table:

Bits	Description
0-2	Microcoded command
3	If set, multiply data portion of command by 16 (allows 16 bit numbers to work, as with the Parallax Basic Stamp product line)
4-31	Data for the associated command

The commands that are currently supported are:

Command	Description
0	Reset pending A/D data to default values
1	Accept new pending data: Code calculates the new rate and scale factors as needed. If the new minimum rate is >= the new maximum rate, then A/D rate control is disabled.
2	Set the pending minimum rate value (default of 1): this is the lowest rate that will be generated when in POT mode
3	Set the pending A/D value that is to be associated with that minimum rate (0 to 4095, default is 0)
4	Set the pending maximum rate value (default of 500,000): this is the maximum rate that will be generated when in POT mode
5	Set the pending A/D value that is to be associated with that maximum rate (0 to 4095, default is 4095)
6	<reserved> Do not use
7	<reserved> Do not use

Note that the system explicitly supports 'reverse slope' mappings: that is to say, you can associate a high A/D reading (such as 4000) with a low rate (10), while a low reading (50) is associated with a high rate (35000). Also, the rates generated by the code are clipped to the values that you specify in subcommands 2 and 4: this allows you to have full control over the actual rates generated by the system.

Please note that you CANNOT generate a rate of 0! That is to say, mapping an A/D reading into 0 with the intent of strictly using the potentiometer to stop the motor will not work. Code in the firmware that operates after the rate conversion code clips the real rate used to the legal limits of the firmware (1 to 500,000).

As is shown in the table, bit 3 may be used to multiply the data-part of the command by 16. This allows controllers which can only generate 16 but unsigned integers to still provide for rates that would otherwise generate too large of a number.

The easiest way to understand the command encoding is that you add the command (0 to 5) to 16 * the data value

As an example sequence, to set the mapping of:

0 volts → 100

3 volts → 2000

you would send the sequence:

0r	Reset all pending values to their defaults
1602r	Set the minimum rate to 100 (16*100 -> 1600)
	The A/D value of 0 is already set as a default value
32004r	Set the maximum rate to 3200 (16 * 2000 → 32000)
59573r	Set the max A/D value to 3 volts (3/3.3 * 4095 → 3723,
	3723* 16 → 59568, 59568 + 5 → 59573)
1r	Accept the new data as the new scalings

You would then actually enable use of the above by setting a target rate of 0:

0R

Also note: the code supports reporting to you the above settings, via the '-16' through '-19' report commands:

-16?	Report A/D based (POT control) min rate value
-17?	Report A/D value at above min rate value
-18?	Report A/D based (POT control) max rate value
-19?	Report A/D value at above max rate value

You should use the above commands when testing your scaling parameters, to make certain that you have correctly encoded your request.

The scripting system commands that implement this function are SetPotRateScales and SetPotRateScalesToDefaults.

R – Set run Rate target speed for vector (SetRunRate)

This defines the run-rate to be used for the next vector to be generated. It may be specified to be between 1 and 250,000 microsteps per second. If a value of 0 is specified, the code resets the value to the default rate of 800. If a value above 250,000 is requested, the code clips it to 250,000.

This defines the equivalent number of microsteps/second which are to be used to run the next vector. The internal vector position is updated at this rate.

As an additional vector motion behavior, if a rate is specified that is < 0 (such as -800R), the vector motion mode becomes "manual chained". In this mode, all automatic ramping of the vectors is disabled. Instead, each vector will have as its target rate the one that you specify, with no slow-down at the end of the vector. It is your responsibility to set your chain of vectors up such that the final one finishes up at a safe stopping speed.

For example,

```
250R
1000X
2000Y
1000G
```

Generates vector from the current location to the target X and Y locations of 1000,2000 with the vector length being treated as 1000 units and the rate for those 1000 units being 250 steps/second.

The power-on/reset default Rate is 800 microsteps/second.

The rate of change of speed is that specified by the 'P' command.

The scripting system command that implements this function is SetRunRate.

r – Restart motion from a pause (‘q’ or limit-switch) state (RequestMotorRestart)

The ‘r’ command is used to restart motion from a pause state caused by the ‘q’ command or through actuation of a limit switch during motion. It is a ‘nested command’, in that it will not abort any pending action; thus, you may restart a paused arc with no bad side effects.

The scripting system command that implements this function is RequestMotorRestart.

$S – Slew asynchronous motors (AsynchGoToLocation, AsynchSlew)

The $S command is used to start the 'slew' action on the currently selected asynchronous motors (see the 'T' and 'm' commands).

If the current value is only "+" or "-" (i.e., just has a sign associated with it), then the motor will slew in the indicated direction on the selected motor(s). Otherwise, the motor(s) will go VALUE steps in the direction indicated by the sign of VALUE, after first stopping the motor (more accurately, the motion will target the current location + x, then act as goto).

For example,

    +$S

will cause the currently selected motors to start slewing in the forward direction, while

    -250$S

will invoke the "relative seek" calculation mode of the firmware.

When doing a relative seek (i.e., "-250$S"), the address calculations are normally based on the current TARGET location, not the current instantaneous location. The actual rules are as follows:

1. If the given motor is currently executing a $GoTo or relative $Seek command, then the new location is calculated as a delta from the old target. For example,

```
Current State:
    Our current location is 1000
    Our current target is 2000
    We are doing a GoTo action
Request:
    -500$S
Calculation:
    Since we are doing a normal GoTo,
    the new target location will be "2000-500", or 1500
Result:
    Motor stops, then goes forward to location 1500
```

2. Otherwise, the current location is treated as the value to calculate from for the relative motion. For example,

```
Current State:
    Our current location is 1000
    We are executing a "+$S" command (slew positive)
Request:
    -500$S
Calculation:
    Since we are executing a Slew,
    the new target location will be "1000-500", or 500
Result:
    Motor stops, then goes backward to location 500
```

This was set up this way as being a reasonable compromise on the intent of the meaning of "relative". If you want to force the motion to be strictly relative to the current location, you issue the "z" (stop) command first. Once that has been issued, the motor is placed in a special state (stopping, no target), which permits relative slew to be from the current location.

For example, to go -500 steps from the current location, regardless of whether the current action is a slew or a targeted goto, issue the command:

    z-500$S

The scripting system command that implements the generic version of "+$S" and "-$S" is AsynchSlew. ***For firmware versions 5.47 and later***, if the current address mode as set by "SetAddressMode" is "Relative", then "AsynchGoToLocation" will use the "$S" command to perform its relative seek action.

S – Select Beginning Arc Angle (DrawArc and DrawSpiral)

‘S’ is used to select the starting angle (in microdegrees) for the ‘Arc’ command. See the ‘A’rc command for more information.

After completion of the ‘A’rc command, the ‘B’egin angle is set to the last angle used in the drawing of the arc.

You will get unreliable operation if you attempt to specify an angle above 359.999999 degrees; i.e., you must always specify a positive start angle less than 360 degrees.

The commands that use the arc function in the scripting system are DrawArc, DrawArcWithRate, DrawSpiral and DrawSpiralWithRate.

T – limit, slew switch control and motor driver enable (SetLimitSwitchEnables)

The limi’T’ switch command is used to control interpretation of the board limit and slew switch inputs, as well as control of the motor output drivers on a per motor basis. The command includes control of whether or not the motor is treated as a "vector" motor or an "asynchronous" motor. **Changing this bit is a critical event** – it must not be changed while there is any motor motion occurring, since the code cannot reconcile vector and asynchronous mode changes while the motor or vector is active.

T' is ignored if you fail to provide a parameter (i.e., 'T' alone results in the board ignoring the command, while '0T' causes the normal 'T' operation with a parameter of '0'). The command takes a bit-encoded parameter, which lists the motor-specific (from the current 'm' selection) actions that are to be performed.

Bits	Action
0-2	Control LIMIT- input actions 2 1 0 0 0 0 (+0) Limit enabled, LOW stops (default) 0 0 1 (+1) Limit enabled, HIGH stops 0 1 0 (+2) Limit instant stop enabled, LOW stops 0 1 1 (+3) Limit instant stop enabled, HIGH stops 1 x x (+4) Limit disabled
3-5	Control LIMIT+ input actions 5 4 3 0 0 0 (+0) Limit enabled, LOW stops (default) 0 0 1 (+8) Limit enabled, HIGH stops 0 1 0 (+16) Limit instant stop enabled, LOW stops 0 1 1 (+24) Limit instant stop enabled, HIGH stops 1 x x (+32) Limit disabled
6	SLEW input disable 0 (+0) Enable slews (default) 1 (+64) Disable slews
7-9	Analog input channel for rate control 9 8 7 0 x x (+0) Analog rate control disabled 1 0 0 (+512) RDY is analog rate source 1 0 1 (+640) NXT is analog rate source 1 1 0 (+768) IO2 is analog rate source 1 1 1 (+896) <reserved>
10	Asynchronous or synchronous (vector) motor 0 (+0) Motor is operated via the VECTOR system 1 (+1024) Motor is operated asynchronously
11	SD6DX only O1 Output signal Manual Control bit 0 (+0) O1 is operated automatically 1 (+2048) O1 is operated via TTL output commands
12	SD6DX only O2 Output signal Manual Control bit 0 (+0) O2 is operated automatically 1 (+4096) O2 is operated via TTL output commands
13	SD6DX only Controls STEP signal polarity 0 (+0) Step is LOW-HIGH-LOW 1 (+8192) Step is HIGH-LOW-HIGH
14	SD6DX only Controls DIR signal polarity 0 (+0) Dir is LOW for minus 1 (+16384) Dir is HIGH for minus
15	BC6D25 only Controls whether a driver fault detected on this motor will abort all motion on the controller. 0 (+0) All motion is aborted on fault 1 (+32768) Do not abort motion on fault
16	Reassign '+' limit input to '-' limit input (allows single-line limit control. If set, the actual I/O port used to control the '+' limit input is reset to the same port as the '-' limit input. The interpretation of that input remains that as specified by bits 3-5 of this command. 0 (+0) +limit input is control of +limit actions 1 (+65536) -limit input is control of +limit actions

As is shown in the above table, the data value is encoded into multiple fields.

Two fields (LIMIT+ and LIMIT- settings) control the limit input behaviors associated with that motor (i.e., whether the limits are enabled at all, the sense level for stopping the motor, and whether the stop is a 'slow-down-and-stop' or an 'instant stop' action).

The next field controls whether the SLEW inputs are monitored for operating the motor when it is in asynchronous mode (slews are always ignored for vector motors).

The next field selects whether analog rate control is enabled for the motor (only when it is an asynchronous motor), and which input channel provides the rate voltage. See the 'v' command for details on configuring the mapping of the voltage input into a real rate.

The next field selects whether the motor is a vector motor or an asynchronous motor. Vector motors are controlled by the vector mode commands, while asynchronous motors are independently controlled by the asynchronous motor commands.

On the SD6DX boards, the following 2 fields control whether the O1 and O2 motor output signals are controlled automatically via the H and N commands, or manually via the TTL output commands.

On the SD6DX boards, the following 2 fields control the polarity of the STEP and DIR output signals.

On the BC6D25 boards, the following field controls whether a “driver fault” will abort all motion (by default, all motion is aborted if a driver chip reports a problem).

By default, this parameter is set to 0. This enables the limit switches for low-true operation, and sets the motor as a 'vector' motor.

On all boards, the next bit controls reassignment of the ‘-’ limit input as control of the ‘+’ limit input actions (single-line limit control). If enabled, this allows the ‘-’ limit input to act as the limit source for both directions of motion, and is most useful for rotational ‘home’ limit actions. Additionally, when this feature is enabled, the associated ‘+’ limit input is effectively ignored for motion control; however, it is still reported in all reports that are appropriate (such as the ‘5?’, ‘6?’ and ‘9?’ reports).

By default, this bit is set to 0 and the feature is disabled.

The command that implements this function in the scripting system is SetLimitSwitchEnables.

t – Configure Special Features (SetIOConfig)

The 't' command is used to configure special features of the RDY and NXT lines, TTL serial operation, and encoder actions. It also is used to configure the range used on control of optional external features, such as the ISODAC system. 't' is ignored if you fail to provide a parameter (i.e., 't' alone results in the board ignoring the command, while '0t' causes the normal 't' operation with a parameter of '0'). It is bit encoded into several groups, each of which control different ports and features.

Bits 0-2: NXT I/O configuration

These 3 bits define the use of the NXT signal

2 1 0	Value	Description
0 0 0	(+0)	NXT is normal TTL/Analog I/O
0 0 1	(+1)	<reserved>
0 1 0	(+2)	NXT is TTL Instant stop, 0 is stop (all motors)
0 1 1	(+3)	NXT is TTL Instant stop, 1 is stop (all motors)
1 0 0	(+4)	NXT is always an analog input, used primarily for motor rate control
1 0 1	(+5)	<reserved>
1 1 0	(+6)	<reserved>
1 1 1	(+7)	<reserved>

Bits 3-5: RDY I/O configuration

These 3 bits define the use of the RDY signal

4 3	Value	Description
0 0	(+0)	RDY is normal TTL/Analog I/O
0 1	(+8)	RDY is fixed as a TTL output, and contains the MOTION COMPLETE flag (1 = motion complete, 0 = busy)
1 0	(+16)	RDY is always an analog input, used primarily for motor rate control
1 1	(+24)	<reserved>

Bits 5: TTL Serial enable

Bit 5 controls whether TTL-Serial is enabled on the board. If enabled, SI2 and SO2 are reserved for use as the TTL-Serial extra I/O lines for daisy-chain serial operation.

5	Value	Description
0	(+0)	Use SI2 and SO2 as generic TTL I/O lines
1	(+32)	SI2 and SO2 are configured for TTL-Serial operation

Bits 6-8: Optional encoder configuration for up to 2 encoders

These 3 bits enable use of the optional encoder alternate operation of some of the slew inputs. When a given encoder is enabled, then the associated TTL lines get redefined as encoder inputs (and their alternate function gets automatically disabled).

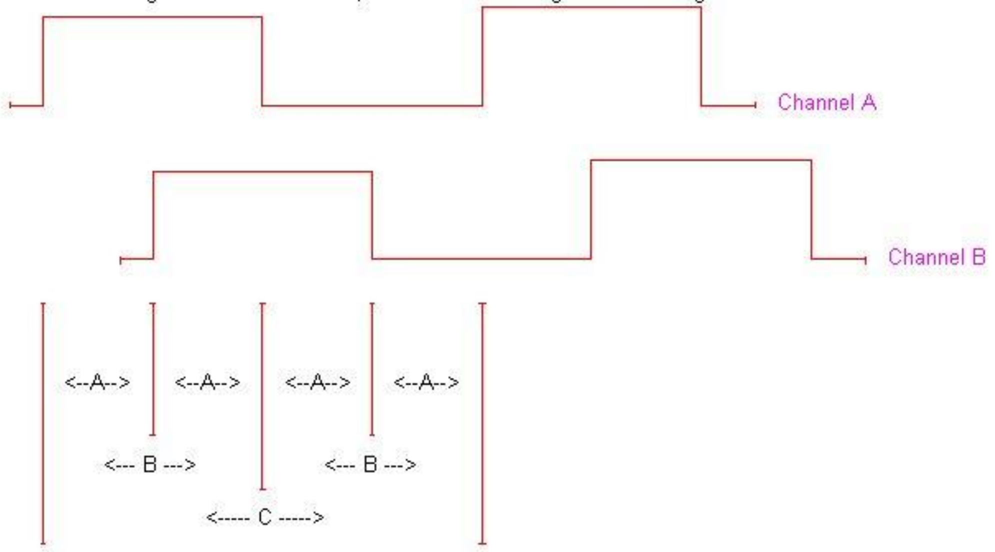
Please note: encoder operation may not be present on all BC6D20 products.

Bit	Value	Signals (A, B)	Description
6	(+64)	LW-, LW+	Enable Limit LW-/LW+ as ENCODER 0 inputs
7	(+128)	RDY, NXT	Enable RDY/NXT as ENCODER 0 inputs (NOTE: If bit 7 is set, then bit 6 is cleared. This means that RDY/NXT selection overrides LW-/LW+ selection as encoder inputs for encoder 0)
8	(+256)	A-, A+	Enable Slew A-/A+ as ENCODER 1 inputs

The encoder operation is a free-running system that monitors the two inputs (for each encoder) for phase-based value changes.

In each case, the code has the following signal requirements:

Timing Diagram for Phase-Encoders
If wired as shown, with 'A' going to A0 and 'B' going to A1, an incrementing count will be generated. If reversed, then a decrementing count will be generated.



Minimum times are:

A: >= x microseconds (minimum time between edges of channel A to B)
B: >= 2x microseconds (minimum pulse width on either channel, up or down)
C: >= 4x microseconds (minum total cycle time on either channel)

If a pulse edge on one signal (such as channel B going low-to-high) occurs within x microseconds of a pulse edge on the other signal (such as channel A going high-to-low), then the associated counter will probably not be updated correctly.

In the current firmware, the minimum sensing cycle time ('A' in the above diagram) is about 1 microsecond.

Bits 9-10: Define the decay mode for the BC6D25 board

These two bits are used to define the current decay mode that is used for the 6 drivers on the BC6D25 board (they are ignored bits on the BC6D20 and SD6DX units).

Bits 9-10	**Value**	**Description**
00	(+0)	Mixed mode decay (the default and recommended setting). Combines FAST and SLOW modes of operation, usually resulting in the best motor operation
01	(+512)	Enable SLOW mode decay only
10	(+1024)	Enable FAST mode decay only
11	(+1536)	(illegal, but treated as Mixed Mode)

In our testing, mixed decay seems to operate most motors the best for many styles of motors, with the FAST mode being best for others. You may wish to experiment to find out which mode is best for your system. The simplest test is to operate at 1/32 of a step as the microstep size, and see which setting works best for your motors.

Bits 11-14: Define the voltage range for optional ISODAC boards

These four bits are used to inform the firmware as to the jumper setting on your external ISODAC boards. The firmware needs to know whether your boards are set up as 0-5 volt (the default) or 0-10 volt systems, in order to correctly scale the output request values in the $J command. If a given bit is CLEAR, then the board jumper is set for 0-5 volt operation. If it is SET, then the board jumper is set for 0-10 volt operation.

If you do NOT correctly configure this feature to match your ISODAC board setting, then your actual DAC voltage will be a factor of 2 off of what it should be!

Bit	**Value if set**	**ISODAC board**
11	(+2048)	ISODAC board 0 jumper set to 10 volts if set, set to 5 volts if clear
12	(+4096)	ISODAC board 1 jumper set to 10 volts if set, set to 5 volts if clear
13	(+8192)	ISODAC board 2 jumper set to 10 volts if set, set to 5 volts if clear
14	(+16384)	ISODAC board 3 jumper set to 10 volts if set, set to 5 volts if clear

The command that implements this function in the scripting system is SetIOConfig.

U – Set TTL Lines low that are '1' in 'x' (QueueSetIOPortsLow, SetIOPortsLow)

'U' is used to set selected bits low on any of the TTL ports that are currently enabled as TTL outputs. It is bit encoded as:

Bit	***Value***	***Signal***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	BO1
28	+268435456	BO2

Add the decimal value for each bit that you want to set to '0' (low), and issue the command on the sum. Those bits will get cleared: the others will be unchanged.

Note that you need to use the 'T' command (T – limit, slew switch control and motor driver enable, SetLimitSwitchEnables) in order to enable access to slew and xO1/xO2 lines, and the 't' command in order to enable access to the RDY, NXT, SI2, SO2 and IO2 lines.

For example, to set the B- line low, you could issue the command:

```
32768U
```

(assuming that B- was configured as a TTL output).

If the value requested is negative (i.e., "-32768U" in the above example), then the request is inserted into the motion queue. It is then executed when that element comes up in the sequence processor, thus allowing a precise setting of a TTL signal when a given vector has completed.

There are several other commands which may be used to monitor and control the output port levels:

- C – Set the TTL lines high that are '1' in 'x'
- D – Set TTL lines to 'x'
- U – Set the TTL lines low that are '1' in 'x'

- u – Set the I/O directions for the slew lines
- The current input values for the limit and slew switches can be retrieved with the '9?' command, as described in the Other report values section.

The scripting functions that implement this command are QueueSetIOPortsLow and SetIOPortsLow.

u - Set TTL I/O directions (SetIOAndSlewPortDirections, SetIOPortDirections)

This command is used to define the I/O directions for all of the programmable I/O ports.

This command is bit-encoded identically to the 'C' command. Any bit with a value of '1' defines the associated port as being an output port. Any bit with a value of '0' defines that port as being an input port

The encoding is therefore:

Bit	***Value***	***Signal: 1 = output, 0 = input***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+

For example, to define NXT on the BC6D20 as an output port, issue the command

```
2u
```

To define W+ and NXT as outputs, you sum the 'values' for NXT and W+ (2 and 64 above, respectively), giving you the command:

```
66u
```

Please note: the low 5 bits of I/O (RDY, POT, SI2 and SO2) must have been defined as 'TTL I/O ports' through use of the 't' command before they can be programmed using the 'u' command.

There are several other commands which may be used to monitor and control the output port levels:

- C – Set the TTL lines high that are '1' in 'x'
- D – Set TTL lines to 'x'
- U – Set the TTL lines low that are '1' in 'x'
- u – Set the I/O directions for the slew lines
- The current input values for the limit and slew switches can be retrieved with the '9?' command, as described in the Other report values section.

The scripting functions that implement this command are SetIOAndSlewPortDirections and SetIOPortDirections.

V – Verbose mode command synchronization (SetVerboseMode)

The 'V'erbose command is used to control whether the board transmits a "<CR><LF>" sequence before it processes a command, and whether a spacing delay is needed before any command response. **By default (after power on and after any reset action), the board is normally configured to echo a carriage-return, line-feed sequence to the host as soon as it recognizes that an incoming character is not part of a numeric value.** This allows host code to fully recognize that a command is being processed; receipt of the <LF> tells it that the command has started, while receipt of the final "*" states that the command has completed processing. 'V' is ignored if you fail to provide a parameter (i.e., 'V' alone results in the board ignoring the command, while '0V' causes the normal 'V' operation with a parameter of '0').

The firmware actually recognizes and responds each new command within a few microseconds of receipt of the stop bit of the received character. In most designs, this will not be a problem; however, since all commands issue an '*' upon completion, and they can also (by default) issue a <CR><LF> pair before starting, it is quite possible to start receiving data pertaining to the command before slower non-interrupt based devices can change their state! In microprocessor, non-buffering designs (such as with the Parallax, Inc.tm Basic Stamp tm series of boards), this can be a significant issue. The firmware handles this via a configurable option in the 'V' command. If enabled, the code will delay 1 millisecond upon receipt of a new command character. This really means that the first data bit of a response to a command will not occur until at least 9 bit intervals after completion of transmission of the stop bit of that command (about 900 uSeconds at 9600 baud); for the Basic Stamptm this is quite sufficient for it to switch from send mode to receive mode.

The verbose command is bit-encoded as follows:

Bit	SumValue	Use When Set
0	+1	Send <CR><LF> at start of processing a new command
1	+2	Delay about 1 millisecond before transmission of first character of any command response.
2	+4	Send numeric responses in hexadecimal instead of decimal.
3-4		<reserved: ignored>
5	+32	If set, code always sends responses: incoming characters do NOT stop pending data. WARNING -- This can cause loss of incoming commands, if 4 or more characters sent while the 64 character transmit buffer is full!
6	+64	If set, sending a 'CTRL-C' character (hex 0x03) will flush any pending input data that has not yet been consumed by the firmware, leaving just the CTRL-C character as the next character to be read. This mode is useful to allow immediate aborts of running scripts.

If you set verbose mode to 0, then the <CR><LF> sequence is not sent. Reports still will have their embedded <CR><LF> between lines of responses; however, the initial <CR><LF> which states that the command has started processing will not occur.

For example,

```
0V
```

would block transmission of the <CR><LF> command synch, and could respond before completion of the last bit of the command, while

```
3V
```

would enable transmission of the <CR><LF>sequence, preceded by a 1-character delay.

The scripting function that implements this command is SetVerboseMode.

W – Set next W ordinate (SetRequestedWLocation)

This command sets the next W value to use on a 'G', 'g', '$A' or '='command.

For example,

```
100W
```

Would cause the next W parameter value to be 100.

The scripting system function that uses this command is SetRequestedWLocation.

X – Set next X ordinate (SetRequestedXLocation)

This command sets the next X value to use on a 'G', 'g', '$A' or '='command.

For example,

```
100X
```

Would cause the next X parameter value to be 100.

The scripting system function that uses this command is SetRequestedXLocation.

Y – Set next Y ordinate (SetRequestedYLocation)

This command sets the next Y value to use on a 'G', 'g', '$A' or '='command.

For example,

```
100Y
```

Would cause the next Y parameter value to be 100.

The scripting system function that uses this command is SetRequestedYLocation.

Z – Set next Z ordinate (SetRequestedWLocation)

This command sets the next Z value to use on a 'G', 'g', '$A' or '='command.

For example,

```
100Z
```

Would cause the next Z parameter value to be 100.

The scripting system function that uses this command is SetRequestedZLocation.

z – Stop motion on currently selected asynchronous motors (AsynchStopMotor)

'z' is used to request a normal slow-down-and-stop action on the currently selected asynchronous motors.

The scripting system function that uses this command is AsynchStopMotor.

= – Define current position for all vector motors (SetCurrentCoordinates)

This command queues a reset of the defined motor locations for all motors. When this point in the motion queue is reached, then the current pending motor values become the new current locations, as opposed to being a new GoTo target.

For example,

```
2000X
4000Y
=
```

Would queue a reset of the current location of the X motor to be 2000, and the current location of the Y motor to be 4000. Note that no actual motor motion is involved – the code simply defines the current location to be that found in the associated motor VALUE register once this queue element is executed.

WARNING! This function changed starting with release 5.48. In versions prior to 5.48, this function affected both vector and asynch motors. As of 5.48 and later, this function just affects vector motors.

This function actually and always redefines ALL vector motor locations based on their current VECTOR based locations. If this function is used, it is best to specify all motor locations for both synchronous and asynchronous motors.

This function also supports the comma separated mode of value input. The order of entry is:

```
W,X,Y,Z,A,B=
```

Parameters that are not specified are actually set to the 'current vector' value, so it is best to specify all of them.

The scripting system function that uses this command is SetCurrentCoordinates.

$= – Define current position for selected asynchronous motors (AsynchSetLocation)

This command resets the current location for the selected asynchronous motors to be the requested value. It also instantly aborts any asynchronous motion on the selected motors.

For example,

```
3m
0$=
```

would reset the W and X motor current locations to be 0, assuming that those motors were set up as being asynchronous motors (see the 'T' command).

The scripting system function that uses this command is AsynchSetLocation.

# – Define current encoder values for the selected motors (SetEncoderCoordinate)

This command may be used to reset one or both encoders to any value that you choose. This function uses the comma-separated method for specifying the encoder values. The first parameter is for Encoder 0, the second is for Encoder 1. If a given encoder is not specified, it not reset.

```
a,b#
```

would set encoder 0 to a, and encoder 1 to b.

```
a#
```

would set encoder 0 to a, and leave encoder 1 unchanged,

```
,b#
```

would just set encoder 1 to b.

The scripting system function that uses this command is SetEncoderCoordinate.

! – RESET – all values cleared, all motors set to "free", redefine microstep. Duplicates Power-On Conditions! (ResetBoard)

This command acts like a power-on reset. It ***IMMEDIATELY*** stops all motors, and clears all values back to their power on defaults. No ramping of any form is done – the stop is immediate, and the motors are left in their "windings disabled" state. This can be used as an emergency stop, although all location information will be lost.

If just '!' is provided without any parameter, the effect is identical to that of issuing the '0!' command, which resets the board to its power-on EEProm-requested settings.

BC6D20/BC6D25 specific:

The value passed is used as the new microstep size, in fixed 1/16th (1/32nd for the BC6D25) of a full step units. **At raw power on, the board acts like a "1!" has been requested on the BC6D20, "2!" on the BC6D25**; that is to say, it sets the microstep size to 1/16th of a step on both the BC6D20 and the BC6D25. By issuing the '!' command, you can redefine the microstep size to a value convenient for your application. The value must range from 1 to 16 (32 on the BC6D25); it is clipped to this range if exceeded.

On the BC6D20, the required values are the powers of 2, vis. 1, 2, 4, 8 and 16 (giving you true microstep step sizes of 1/16, 1/8, ¼, ½ and 1 respectively). Similarly, on the BC6D25, the required values are the powers of 2, vis. 1, 2, 4, 8, 16 and 32 (giving you true microstep step sizes of 1/32, 1/16, 1/8, ¼, ½ and 1 respectively). All other values (such as RATE or GOTO LOCATION) are then expressed in units of the microstep size; therefore, location "3" would mean "3/16" in the 1/16h step resolution (microstep set to 1 on the BC6D20 or to 2 on the BC6D25), and "3" in the largest resolution (microstep set to 16 on the BC6D20 or 32 on the BC6D25).

For example,

```
1!
```

resets the BC6D20 system to its power on default of 1/16 microstep resolution.

SD6DX specific:

The value passed is used as the new step pulse width, in 1 microsecond units. **At raw power on, the board acts like a "2!" has been requested**; that is to say, it sets the pulse width to 2 microseconds. By issuing the '!' command, you can redefine the step pulse width to match the requirements of your step-and-direction driver board. The value must range from 1 to 100; it is clipped to this range if exceeded.

For example,

```
2!
```

resets the system to its power on default of 2 microseconds pulse width.

Special Values

There are 2 special values reserved for the '!' reset command:

0! – Reboot using the EEProm-specified settings, including the microstep size

-1! – Reboot using the factory default settings

The reset command also resets all other settings to their EEProm-specified values: By default, these are:

- **BC6D20: 1!** – Set the microstep size to 1/16 of a step per logical step
- **BC6D25: 2!** – Set the microstep size to 1/16 of a step per logical step
- **SD6DX: 2!** – Set the step pulse width to 2 microseconds
- **9600b** – Set the communications baud rate to 9600 baud
- **0**= – Reset all motors to be at location 0

- **0#** – Reset both encoders to be at location 0
- **BC6D20/BC6D25: 500H** – Set motors to 0.5 amps when moving
- **SD6DX: 1H** – Set O1 high, O2 low when motors are moving
- **80K** – Set the "Stop OK" rate to 80 microsteps/second
- **8000P** – Set the rate of changing the motor speed to 8000 microsteps/second/second
- **800R** – Set the target run rate for the motor to 800 microsteps/second
- **0T** – Enable all limit switch detection
- **0t** – Set all I/O ports to our defined default settings
- **1V** – Set <CR><LF> sent at start of new command, no transmission delay time
- **BC6D20/BC6D25: 0N** –Motor windings are off when idle
- **SD6DX: 2N** –Set O1 low, O2 high when motors are idle

The scripting system function that uses this command is ResetBoard.

[- Specify text parameter (SetTextField), or enter script mode

The '[' command is used to specify text fields. The 'x' parameter specified before the '[' defines the type of text being presented, while all text after '[' is copied into the appropriate field until a <CR> character (decimal value 13) is seen.

The '[' values that are currently defined are:

0[text…	Simple echo back of whatever is in text, for software synch
1[text…	Specify user startup commands
2[…	Enter command-line scripting mode. All future text input is handled by the script system, until told to exit.
3[…	Send one command to the scripting system. Allows execution of any script command with automatic return to 'normal' operation

The function reports back what it has accepted when the 0 and 1 modes are used, in the form of

```
[,#,text…
```

For example, to specify startup text to set the start rate for vectors, you could issue the command

```
1[2000R
```

The code would echo back

```
[,1,2000R
```

Please see the later section entitled "The Scripting System" for documentation about the scripting system that is available via the "2[" and "3[" commands.

Assuming that normal verbose communications is enabled, the sequence would appear as follows on a terminal emulator (such as SimpleSerial):

1[	You send the '1['
2000R	The board echoes <CR><LF>, you send '2000R <CR>'
[,1,2000R	The board echoes the complete result back to you
*	The board sends back the command completion '*' response

The startup command is NOT saved into the EEProm until you issue the "memorize settings" command "-123456789e". This means that if you do a reset board before issuing the memorize command after having set a new startup string, the new startup string is ignored.

Use of the '0[text…' command can be useful for resynchronizing with the board if you have sent multiple commands and do not want to count '*' being returned by the board. Just append the '0[text…<CR>' stream to your command list, and look for the matching '[,0,text…' return text. The * following that text is fully synchronized with your commands up to that point.

The '1[' mode may be used to define startup conditions that are "clearer" than those generated by "just" saving the current state. Since that mode allows you to enter an arbitrary set of commands that get executed at startup, it is more obvious as to what is being requested.

Please note: if "[" is seen as part of the startup command list, all text after the "[" will get sent to the scripting language parser as a command to be performed AFTER the rest of startup has fully completed. Please see the "The Scripting System" section for more details.

? – Report status (ReportNumericSingleValue, ReportNumericSingleValueForMotor)

The "Report Status" command ("?") can be used to extract detailed information about the status of either motor, or about internal states of the software.

For a status report, the value is interpreted as from one of three groups:

1-255: Report memory location 1-255

Useful locations: ***NOTE THAT ANY LOCATION ABOVE 7 MAY CHANGE BETWEEN CODE VERSIONS***

- 5: Report motor-specific elements
- 6: Report all of the limit inputs
- 7: Report all of the slew inputs , plus extended TTL data
- 8: Report the extended IO ports
- 9: Report combined IO and Slew ports

0: Report items –1 through –11 from the following special reports

<0: Report special item as listed later in this section: report a value of 0 if the report request is illegal.

All of the reports follow a common format, of:

1. If Verbose Mode is on, then a <carriage return><line feed> ("crlf") pair is sent.
2. The "M" followed by a digit corresponding to the motor being reported on is sent (i.e., 'M2' for X, or "M3" for Y).
3. A comma is sent.
4. The report number is sent (such as –4, for target position).
5. Another comma is sent.
6. The requested value is reported.
7. If this is a report for multiple motors, then a <crlf> is sent.
8. If this is a report for multiple motors, another report is sent for each motor.
9. If Verbose Mode is on, then a <crlf> is sent
10. A "*" character is sent.

If all motors are being reported, a line for each motor is sent.

Finally, a "*" character is sent, which notifies the caller that the report is complete.

Note that in the following examples, first line of "Received" is "*". This is because two commands are actually being sent (i.e., "B", then "-<whatever>?"), and each command always generates a "*" response once it has been completed. Technically, fully "synchronized" serial communication consists of (1) send a command, and (2) save all characters until the "*" response is seen. The intervening characters are the results of the command, although only report ("?") and reset ("!") generate any significant response.

The special reports which are available are as follows:

0: Report all common reportable items

The "report all common reportable items" mode reports the data as a comma separated list of values, for reports –1 through –11. Just after power on, for example, the request of "0?" would generate the report:

Mx,0,a,b,c,d,e,f,g,h,I,j,k,l,m

Where:

- Mx is the motor:
  - M1: W
  - M2: X
  - M3: Y
  - M4: Z
  - M5: A
  - M6: B
- 0 is the report number; 0 is the 'all' report
- a is the value for the current location (report "-1")
- b is the value for the current speed (report "-2")
- c is the value for the current slope (report "-3")
- d is the value for the target position (report "-4")
- e is the value for the target speed (report "-5")
- f is the value for the windings state (report "-6")
- g is the value for the stop windings state (report "-7")
- h is the value for the step action (motor state) (report "-8")
- i is the value for the step style (both full step modes and half) (report "-9")
- j is the run rate (report "-10")
- k is the stop rate (report "-11")

For example,

```
6m0?
```

Would report all reportable values for the X and Y motors. You could receive:

```
*
M2,0,30,10,1000,30,10,0,0,0,1,100,10
M3,0,-300,10,1000,-300,10,0,0,0,1,100,10
*
```

-1: Report current location (ReportCurrentLocation)

This reports the current (instantaneous) location for the selected motor(s).

For example,

```
6m-1?
```

Would report the current location on the X and Y motors. You could receive:

```
*
M2,-1,10
M3,-1,25443
*
```

The script system implements this as ReportCurrentLocation.

-2: Report current speed

This reports the current (instantaneous) speed for the selected motor(s).

For example,

```
6m-2?
```

Would report the current speed on the X and Y motors. You could receive:

```
*
M2,-2,800
M3,-2,2502
*
```

-3: Report current slope

This reports the current (instantaneous) rate of changing the speed for the selected motor(s).

For example,

```
6m-3?
```

Would report the current rate on the X and Y motors. You could receive:

```
*
M2,-3,10
M3,-3,25443
*
```

-4: Report target position

This reports the target location for the selected motor(s).

For example,

```
6m-4?
```

Would report the current target on the X and Y motors. You could receive:

```
*
M2,-4,100
M3,-4,-35443
*
```

-5: Report target speed

This reports the current target run rate which is desired for the selected motor(s). This value is usually either the current stop rate (we are attempting to slow down to this speed) or the current requested run rate (as reported by –10, and as requested by the 'R' command) depending on whether we are speeding up or slowing down.

For example,

```
6m-5?
```

Would report the target rate on the X and Y motors. You could receive:

```
*
M2,-5,800
M3,-5,250
*
```

-6: Report windings state

This reports the current energized or de-energized state for the windings for the selected motor(s). A reported value of 0 means "the windings are off", a value of 1 means "the windings are energized in some fashion".

For example,

```
6m-6?
```

Would report the current state on the X and Y motors. You could receive:

```
*
M2,-6,1
M3,-6,0
*
```

-7: Report stop windings state

This reports whether the windings will be left energized when motion completes for selected motor(s). A reported value of 0 means "the windings will be turned off", a reported value of 1 means "the windings will be left at least partway on".

For example,

```
6m-1?
```

Would report the requested state on the X and Y motors. You could receive:

```
*
M2,-1,1
M3,-1,0
*
```

-8: Report current step action (i.e., motor state)

This reports the current (instantaneous) state for the selected motor(s). The step action may be one of the following values in the version 5.11 firmware, although this can change in future releases:

- 0: Idle; all motion complete (note: this will always be true)
- 1: motor in motion; either ramping up or at target speed
- 2: ramping down; either due to speed variation request or due to a stop request.
- 3: Slewing ("+$S")
- 4: Ramping down on a quick stop action
- 5: reversing direction of a slew
- 6: New goto detected: doing extended motor actions to make the transition as smooth as possible
- 7: Vector motion is active on this motor
- 8: If motor is in vector mode, this value will normally appear if this motor has no motion during this vector

For example,

```
6M
-8?
```

Would report the current state on the X and Y motors. You could receive:

```
*
M2,-8,0
M3,-8,3
*
```

This would mean that motor X is idle, while motor Y is currently doing some form of slew operation.

The script system implements this as ReportCurrentStepAction

-9: Report step style (i.e., micro step, half, full)

This reports the current method of stepping for the selected motor(s). The only step style reported is:

- 3: Microstep

For example,

```
6m-9?
```

Would report the current stepping method on the X and Y motors. You could receive:

```
*
M2,-9,3
M3,-9,3
*
```

This would equate to the X motor being in microstep mode, while the Y motor is running in full-power, full step mode.

-10: Report run rate

This reports the current requested run rate for the selected motor(s). This is the last value set by the "R" command.

For example,

```
6m-10?
```

Would report the current rate on the X and Y motors. You could receive:

```
*
M2,-10,2000
M3,-10,3200
```

*

-11: Report stop rate

This reports the speed at which the motors may be considered to be stopped, for starting and stopping activities for the selected motor(s).

For example,

```
6m-11?
```

Would report the current stop rate on the X and Y motors. You could receive:

```
*
M2,-11,80
M3,-11,50
*
```

-12: Report current software version and copyright (ReportCopyRightText, ReportFirmwareVersion)

This reports the software version and copyright.

For example,

```
6m-12?
```

could report:

```
*
M2,-12
BC6D20NCRouter 5.11 May 4, 2015
Copyright 2014, 2015 by Peter Norberg Consulting, Inc. All Rights
Reserved.
*
```

The script system implements as ReportCopyRightText and ReportFirmwareVersion.

-13: Report I/O configuration (‘t’ command data) (ReportIOConfig)

This reports the I/O configuration data as used by the “t” command (t – Configure all I/O port directions and use).

For example,

```
2m
-13?
```

Would report the current I/O configuration. You could receive:

```
*
M2,-13,0
*
```

The script system implements this as ReportIOConfig.

-14: Report microstep size (ReportMicrostepSize)

This reports the current microstep size (parameter for the ‘!’ command)

For example,

```
2m
-14?
```

Would report the current microstep size for the selected motor. You could receive:

```
*
M2,-14,4
*
```

The script system implements this as ReportMicrostepSize.

-15: Report the maximum step rate supported by this firmware

This reports the maximum step rate (in microsteps/second) supported by this firmware release.

For example,

```
2m
-15?
```

Would report the maximum rate. You could receive:

```
*
M2,-15,250000
*
```

-16: Report A/D based (POT control) minimum rate value

This reports the minimum speed which may be attained using the A/D rate mode of control for this motor.

For example,

```
6m
-16?
```

Would report the minimum POT rates on both the X and Y motors. You could receive:

```
*
M2,-16,1
M3,-16,20
*
```

-17: Report A/D setting at minimum rate (ReportAToDAtMinPotRate)

This reports the A/D setting at the minimum speed which may be attained using the A/D rate mode of control for this motor.

For example,

```
6m
-17?
```

Would report the A/D settings at the minimum POT rates on the X and Y motors. You could receive:

```
*
M2,-17,0
M3,-17,4095
*
```

The scripting system implements this as ReportAToDAtMinPotRate.

-18: Report A/D based (POT control) maximum rate value

This reports the maximum speed which may be attained using the A/D rate mode of control for this motor.

For example,

```
2m
-18?
```

Would report the maximum POT rates on the X motor. You could receive:

```
*
M2,-18,13500
*
```

-19: Report A/D setting at maximum rate (ReportAToDAtMaxPotRate)

This reports the A/D setting at the minimum speed which may be attained using the A/D rate mode of control for this motor.

For example,

```
6m
-19?
```

Would report the A/D settings at the maximum POT rates on the X and Y motors. You could receive:

```
*
M2,-19,4095
M3,-19,0
*
```

The scripting system implements this as ReportAToDAtMaxPotRate.

-23: Report Maximum Allowed Current Setting (mA)

This reports the maximum allowed current setting for motors connected to the board, expressed in milliamps (mA).

For example, with

```
2m
-23?
```

You could receive:

```
*
M2,-23,2000
*
```

```
which would mean that your board is configured for 2 amp operation.
```

-24: Report current 'o' scale factor as a fixed point number

This reports the current 'o' setting used to scale 'H' and 'N' values into dac values for the motors.

For example, with

```
2m
-24?
```

You could receive:

```
*
M2,-24,0.1235
*
```

Note: '0.1235' is the value that is reported as the default scale factor for millamp-based scaling.

-25: Report 'H' dac value (current request when motor enabled)

On the BC6D20 and BC6D25, this reports both the current 'H' setting and the raw dac value

For example, with

```
2m
-25?
```

You could receive:

```
*
M2,-25,502,62
*
```

In the above example, the '502' is the 500 mA request, while the '62' shows the actual dac value that is generated.

On the SD6DX, this reports the raw “H” value.

-26: Report 'N' dac value (current request when motor idle)

On the BC6D20 and BC6D25, this reports both the current 'N' setting and the raw dac value

For example, with

```
2m
-26?
```

You could receive:

```
*
M2,-26,97,12
*
```

In the above example, the '97' is the 100 mA request, while the '12' shows the actual dac value that is generated.

On the SD6DX, this reports the raw “N” value.

-27: Report the manufacturing startup command list

This reports the manufacturing startup command string. This is a special order factory series of commands that are executed as part of the startup sequence every time that the board is reset.

For example, with

```
2m
-27?
```

You could receive:

```
*
M2,-27,2000R
*
```

-28: Report current backlash settings

This reports the current backlash setting for the motor.

For example, with

```
2m
-28?
```

You could receive:

```
*
M2,-28,2000
*
```

```
which would mean that the X motor on your board is configured
for with a backlash setting of 2000.
```

-29: Report the copyright extension text

This reports the copyright extension text string. This is a special order element that gets appending to the firmware name in the '12?' query.

For example, with

```
2m
-29?
```

You could receive:

```
*
M2,-29,SpecialOrderA
*
```

-30: Report the user startup command list

This reports the user startup command string. This is a user specified series of commands (from the '0[' command) that are executed as part of the startup sequence every time that the board is reset.

For example, with

```
2m
-30?
```

You could receive:

```
*
M2,-30,2000R
*
```

-31: Report Encoder Values (ReportCurrentEncoderLocation)

This reports the current encoder values.

For the purposes of reporting the two encoders, motor 'W' is treated as encoder 0, while motor 'X' is treated as encoder 1.

For example, with

```
2m
-30?
```

You could receive:

```
*
M2,-31,12415
*
```

The script system implements this as ReportCurrentEncoderLocation.

-32: Report Limit Switch Enables (ReportLimitSwitchEnables)

This reports the limit switch enables for the given motor, as requested by the ‘T’ command.

The script system implements this as ReportLimitSwitchEnables.

-33: Report the I/O port direction settings (ReportIOPortDirections)

This reports the I/O port direction settings, as defined by the ‘u’ command.

The script system implements this as ReportIOPortDirections.

Other report values

5: Reports for the selected motor (ReportMotorSwitches)

This is used to provide a snapshot of the up to 6 TTL input lines associated with the current motor (M).

Bit	Value	Description
0	+1	LM- (motor '-' limit)
1	+2	LM+ (motor '+' limit)
2	+4	M- (motor '-' slew)
3	+8	M+ (motor '+' slew)
4	+16	O1 (motor O1 output, SD6DX)
5	+32	O2 (motor O2 output, SD6DX)

The script system implements this as ReportMotorSwitches.

6: Report all LIMIT inputs (ReportLimitSwitches)

This is used to provide a snapshot of the LIMIT SWITCH input values. The report value is bit mapped as follows:

Bit	Value	Description
0	+1	LW- (W- limit)
1	+2	LW+ (W+ limit)
2	+4	LX- (X- limit)
3	+8	LX+ (X- limit)
4	+16	LY- (Y- limit)
5	+32	LY+ (Y+ limit)
6	+64	LZ- (Z- limit)
7	+128	LZ+ (Z- limit)
8	+256	LA- (A- limit)
9	+512	LA+ (A+ limit)
10	+1024	LB- (B- limit)
11	+2048	LB+ (B+ limit)

The script system implements this as ReportLimitSwitches.

7: Report SLEW inputs (ReportSlewSwitches)

This is used to provide a snapshot of the SLEW and IO port input values. The report value is bit mapped as follows:

Bit	Value	Description
0	+1	W- (W- slew)
1	+2	W+ (W+ slew)
2	+4	X- (X- slew)
3	+8	X+ (X- slew)
4	+16	Y- (Y- slew)
5	+32	Y+ (Y+ slew)
6	+64	Z- (Z- slew)
7	+128	Z+ (Z+ slew)
8	+256	A- (A- slew)
9	+512	A+ (A+ slew)
10	+1024	B- (B- slew)
11	+2048	B+ (B+ slew)

The script system implements this as ReportSlewSwitches.

8: Report extended IO port inputs (ReportExtendedInputs)

This is used to provide a snapshot IO port input values. The report value is bit mapped as follows:

Bit	Value	Description
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2

The script system implements this as ReportExtendedInputs.

9: Report combined IO and SLEW port inputs, matching the TTL I/O function format (ReportTTLValues)

This is used to provide a snapshot IO port and slew input values. The report value is mapped identically to the TTL I/O commands (such as 'C' and 'D')

Bit	*Value*	*Signal*
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	BO1
28	+268435456	BO2

The script system implements this as ReportTTLValues.

16384 to 16415: Report User EEProm data

This report echoes back the contents of the user data within the EEProm. As is described in the documentation for the 'e' command, there are 32 user-data words available. A specific side effect of this report is to set the next EEProm write address to refer to the data word that was just reported.

User data word 0 is mapped into report value 16384 (i.e., add 16384 to the user data element ID to determine the report value to use to extract the data).

The scripting system implements the equivalent of this request in the EEPromReadData function (with parameter selection being 0 to 31 instead of 16384 to 16415).

16416 to 16447: Report fixed firmware configuration data

This report provides information from the fixed configuration region of the board.

Value	ID	Description
16416	0	Board type
16417	1	Board serial number within manufacturing run
16418	2	Manufacturing run number for board
16419	3	Manufacturing reference number for board
16420	4	Special order firmware flags
16421-16447	5-31	Reserved for future expansion

Board type: This describes the type of board. As of July 18, 2016, the current board types for StepperBoard products are:

```
0: BS1010, SS1010 series
1: BC2D20 series
2: BC4E20 series
3: SD4DX series
4: FENC series
5: BC6D20 (TI) series
6: BC6D25 (TI) series
7: SD6DX (TI) series
```

Board serial number within run: unique serial number within manufacturing run for this board.

Manufacturing run number: Board run encoded information. This number, combined with the board serial number, provides a unique board identifier. The manufacturing run number is encoded as:

Bits 0-15: Raw run number

Bits 16-23: Manufacturing process code.
If this 8-bit code may be represented as an upper case letter (i.e., if it has numeric value of 65 through 90), then it is reported that way. Otherwise, it is reported as a 2-digit hexadecimal value.

The serial number/manufacturing run number pair are usually reported as:

LRRR-SSS

```
Where L is the process code, RRR is the run number, and SSS is the
serial number.
```

other – Ignore, except as "complete value here"

Any invalid command is simply ignored, other than sending a response of "*". However, if a numeric input was under way, that value will be treated as complete. For example,

123 456G

would actually request a "GoTo location 456". Since the " " command is invalid, it is ignored; however, it terminates interpretation of the value which had been started as 123.

Note that, upon completion of ANY command (including the 'ignored' commands), the board sends the <carriage return><line feed> pair, followed by the "*" character (as configured by the 'V' command).

Scripting System

The BC6D20/BC6D25/SD6DX firmwares share a common scripting system, that is very loosely modeled on a BASIC-like syntax system, and is accessed through the "2[" and "3[" commands. Those commands change the input system to be line oriented, waiting for a <CR> before any action is performed. Additionally, processing of a CTRL-C character (hex 03) as an input character to the script will either abort the script (if running a script) or exit out of the scripting system (if in the command line mode requested via the "2[" command).

"2[" places the firmware into a command-line mode, that allows you to use a simple terminal emulator (such as the SimpleSerial application) to type in and run commands and programs. It remains in that mode until the user (or program) either exits via the "EXIT" command, types a CTRL-C while entering commands, or if the board is repowered or reset.

"3[" executes exactly one command (all text up to the following <CR>), and then returns to the normal serial command mode described by the rest of this manual. The commands are identical to those entered in the "2[" mode of operation; the only difference being that the system immediately returns to the normal serial command mode as soon as the command completes, instead of waiting for more scripting commands.

The script system **always** monitors the primary serial input stream (port 0) for a CTRL-C character as being the next character to process. If it sees that character, it will abort any executing script. The 'SetVerboseMode' function (the 'V' command in the non-scripting system) also has a control bit that tells the low level serial buffering system to automatically purge any unprocessed serial characters when a Ctrl-C is seen, so that an incoming CTRL-C becomes the next character to be processed, period.

A Simple Example Script

Before going into the details of the language, it may be instructive to run a small sample program.

1. Start up your controller, and connect to it via the "SimpleSerial" application
2. Type the command

   ```
   2[
   ```

3. To place it into command-line script mode
4. Type in the following simple program

   ```
   10 ' Simple program to show some powers of 2
   20 A=1
   30 FOR I=0 TO 30
   40 PRINT I, " ", A
   50 PRINT
   60 A=A*2
   70 NEXT
   80 END
   ```

5. Tell the program to run

   ```
   RUN
   ```

6. You should see a simple powers of 2 table printed, followed by the text "READY"

The main items of note should be:

1. If a line starts with a number, then it gets inserted into the current program sorted by that number.

2. If a line does not start with a number, then the line is immediately executed as a command.
3. The "2[" mode of operation always prints "READY" when it is ready for a new command to be entered (which also means that a script is not currently running).
4. The Ctrl-C character may be typed to abort a running program IF that program requests serial input. If a program does not request serial input, the program can only be stopped by an error in the program, a feature in the program (where it hits an "END" or "EXIT" command), or by restarting the board (RESET or power cycle)
5. Some commands are designed to be used from the command line. However, there is no built-in limit on that behavior: ALL commands may be part of a program.
6. The language supports variables, constants, some common mathematical and string operations, and complete access to all features of the board supported by the normal serial system via commands that are listed with each serial command (for example, the "!" command is supported via the "ResetBoard(…)" function call).
7. The program is tokenized as it is entered, with the entire program being set up for immediate execution as soon as it has been typed in.
8. All commands and variable names are case insensitive. When a program is listed back via the "LIST" command, it is presented in upper case.

Basic Syntax

The scripting system supports a simple BASIC-like syntax. Lines (commands) consist of an optional line number and one command. Each command is either a variable assignment or an extended command from the following list:

```
IF <expression> THEN <expression> {ELSE <expression>}

FOR <var>=<expression> TO <expression> {STEP <expression>}
NEXT

GOSUB <expression>
RETURN

GOTO <expression>

PRINT <expression list>

LIST {<optional expression 1> {<optional expression 2>}}

TRACE <expression>

RUN

END

EXIT

NEW
LOAD
SAVE
```

Additionally, it is legal to just list a function name without an assignment (as a simplified method of calling the motion control functions without worrying about the return value).

Expressions are a mixture of operations using operators (such as "+" and "-"), variables, numeric constants, string constants and function calls. The code automatically converts between expression types (integer, floating point, character and string) based on the semantics of the expression at that point in time.

Constants

Numeric constants support integer, floating point and exponential notation. All values are internally expressed as signed 32 bit values, so the integer range is -2,147,483,649 through 2,147,483,647. The floating point range limits are approximately +/- 1.0E37.

String constants consist of text enclosed by double quotes. There a few "special" escaped characters that are used to allow easier expression of otherwise awkward values:

Sequence	Use
\a	0x07: Bell
\b	0x08: Backspace
\f	0x0C: Top of form
\n	0x0A: Line feed
\r	0x0D: Carriage Return
\t	0x09: Tab
\v	0x0B: Vertical tab
\"	": Single quote
\\\\	\\: backslash
\<anything else>	Allow any character unchanged

Thus, "\a" would generate a string one character long, consisting of the bell character (hex value 7).

Line Numbers

A line number is an integer from 1 to 1000000000 (1 billion) that starts a line. This number is used to both sort the line in numerical order, and to identify the line as part of a program as opposed to being an immediate command.

For example

    10 PRINT "HI THERE\r\n"

Would create a line (number 10), that has the command "PRINT HI THERE\r\n" as being the statement to be executed. On the other hand, just entering

    PRINT "HI THERE\r\n"

Would cause the script system to print

    HI THERE

Immediately.

If you enter a line number that already exists, the prior line is replaced with the new line.

Comments

A comment may occur on any line, and is indicated by a single quote character ('). From that point on, the rest of the line is ignored by the parsing system.

For example,

```
10 ' This is a comment-only line that is saved with the program
20 A=25 ' While this comment can explain the assignment (A=25)
```

Variables

Variables within the scripting system are probably the most confusing feature, with the greatest likelihood of causing foot damage (as in "shooting yourself in your foot...") when improperly used.

A variable consists of:

- A letter A through Z
- An optional digit 0 through 9
- An optional variable type:
    - '#' defines a floating point variable
    - '$' defines a string variable
    - Nothing means an integer variable
- An optional subscript, consisting of a "(", followed by a numeric expression, followed by a ")"

For example:

```
A       Variable
A0      Integer Variable A0
A1#(1)  Floating point array element 1 (counting from 0), relative to A1
A$      String variable A
```

The confusing aspect comes in when you learn that all variables are pulled from a fixed array of 260 32-bit elements, with the primary variable name ("A", "B" etc) describing where the variable occurs within that array.

A starts at array element 0, B at element 10, C at element 20, and so on. This means that EVERY variable is really just a way of getting convenient named access to the array, and is just a base address. A0 is really the same as A, which is the same as A(0). B maps into B0, B(0), and A(10).

Adding the variable type qualifier ("#" and "$") just tells the code how to access that array element. If there is no qualifier, the element is treated as a 32 bit integer. If "#" is used, then the element is treated as a 32 bit floating point, while if "$" is used, then the element is treated (mostly) as a null-terminated string.

Since the values explicitly overlap, it is quite easy to have some hard-to-trace bugs. For example,

A=45

PRINT A#

Will print 6.305845E-44 instead of 45. Since you are directly accessing the raw memory that contains the value, no conversion is done.

However, if you then do

A#=A

PRINT A#

You will get the expected value of 45. Expressions are automatically converted as needed.

Note also that array access is relative to the array identifier from the base variable-with-digit notation. The following terms reference the same value:

B(3)

B3

B0(3)

B1(2)

B2(1)

Also, on a less "obvious" scale,

A9(4)

A(13)

And so on.

The other killer is strings: a string is a sequence of characters terminated by a character with a binary value of 0 (called a "Null"). 4 characters are packed into one 32 bit memory location, so the string "01234567" will actually occupy at least part of 3 of the array locations. If we had A$="01234567", then A(0) would contain the characters "0123", A(1) would contain "4567", and A(2) would have a "0" at the next sequential memory location (with the remaining 3 characters undefined).

The only limit on the length of a string is the end of the main 260 element array: a string cannot go beyond that end (enforced by the script system), and there is an extra "0" element at the end of that array to force any string to terminate there.

This means that the maximum length available to string A$ is 4*260, or 1040 characters. However, the maximum length available to Z$ is 40 characters, due to its location in the array.

The following demonstrates some of the other details on string processing:

```
10 A$ = "This is a test"
20 FOR I = 0 TO 14
30 PRINT "A$(" , I , ")=" , A$ ( I ) , " "
40 J = A$ ( I )
50 PRINT "j=" , J
60 PRINT
70 NEXT
80 END
```

When run, it generates:

```
A$(0)=T j=84
A$(1)=h j=104
A$(2)=i j=105
A$(3)=s j=115
A$(4)=  j=32
A$(5)=i j=105
A$(6)=s j=115
A$(7)=  j=32
A$(8)=a j=97
A$(9)=  j=32
A$(10)=t j=116
A$(11)=e j=101
A$(12)=s j=115
A$(13)=t j=116
A$(14)= j=0
```

You will observe that the print command treats a single byte from a string as a single character for printout, but that assigning that byte to an integer variable lets it be treated as an integer.

Common expressions

The "basic" language consists of commands that normally contain multiple expressions. The expressions look like normal mathematical statements, which consist of variables, constants and functions separated by operators.

Functions are reserved keywords; if the function needs parameters (such as "ResetBoard(1)"), then the keyword includes a trailing left parenthesis, which you then follow by a comma separated list of expressions, which is then terminated by a right parenthesis.

The operators are used to perform mathematical or logical operations on the associated expressions. They are divided up into unary and binary operators, the separator ",", and the grouping mechanism provided by the "(" and ")" parenthesis operation. In general, operations are performed left-to-right, with overrides based on the precedence of the operator being used.

Expressions within parentheses are processed before other expressions. Then, in order of priority, we have:

Unary operators are processed just after parentheticals

Unary operators are Unary '+', Unary '-' and '~' (logical NOT).

- Unary '+' is mostly ignored: it just states that the trailing expression's sign is not changed by the operation.
- Unary '-' is used to change the sign of the immediately following expression (such as "-3").
- '~' performs a ones-complement of the following value. A 0 becomes a -1, and vice-versa. Since this is a bit-wise complement, this is a convenient tool for generating masks.

Binary operators are processed in the most common order

Binary operators use the most common precedence for the order of operations. In order of processing (highest first), we have:

- Multiplication '*' and Division ('/') are done first
- Then addition '+' and subtraction '-'
- Followed by the logical and comparison operators:
  - & (AND)
  - | (Inclusive Or)
  - ^ (Exclusive Or)
  - << (Logical shift left)
  - >> (Logical shift right)
  - <> (Not Equals compare)
  - < (Less than compare)
  - = (Equals compare)
  - > (Greater than compare)
  - <= (Less than or equal compare)
  - >= (Greater than or equal compare)

Comma operators are lowest priority

The comma operator (",") is processed last, and is grouped from left-to-right (so that comma-separated printouts occur in the correct order).

Basic Commands

The language consists of just a few basic commands, with all of the motor-specific functionality being provided by a large list of functions that may be used in expressions.

All of the commands may be entered with or without a leading line number. If entered without a line number, they are treated as immediate mode commands, and are just executed as requested. A few of the commands (FOR…NEXT and RETURN) do not make any sense when executed from the command line, and will usually generate run-time errors due to their not having a valid context for operation.

NEW clears the program from memory

NEW will fully clear the current program and variables. If it is part of a program, it will stop the programs' operation (since the program is now gone).

RUN executes the program

RUN is used to actually execute a program. It will (re) start the program at the beginning, and will clear out any stacks (such as the gosub and for…next stacks) before beginning execution. It also explicitly clears the variables: the variable array is set to all 0 as part of a RUN request.

If RUN is executed within a program, the effect is the same as restarting the program from the beginning, with all variables set to 0.

SAVE will save the current program to FLASH memory

SAVE is normally just used from the command line to save the current program into flash memory. Note that the variables array is not saved, and any prior program in flash memory is erased.

LOAD will load the previously SAVEd program from FLASH memory

LOAD will first run the NEW command to clear the current program and variables, and will then copy the previously SAVEd program from FLASH memory into the script region. The program will not be started: It will just be available to run.

END terminates the program

END is used to terminate the program, and return it to the command-line environment.

If the scripting system was started via the "2[" command, then you will see a "READY" prompt printed, and the scripting system will wait for further commands to come in on the serial system.

If the scripting system was started via the "3[" command, then the firmware will return to the common single-character command mode, and send an "*" through the serial port to indicate that it is ready for more commands.

You may have as many "END" statements in your program as you want.

EXIT terminates the program and exits the scripting system

EXIT is a more specific form of END, that strictly exits the scripting environment and returns to the single-character command mode. It sends an "*" through the serial port to indicate that the firmware is ready for more single-character commands.

LIST {<line1> {<line 2>}}

The LIST command prints a listing of the current program. If <line1> is specified, then just <line1> is shown. If both <line1> and <line2> are specified, then the range of lines from <line1> through <line2> are shown.

TRACE <expression> is used to trace program execution

The TRACE command is a debug tool that allows you to view exactly how your program is executing. By default, TRACE is set to 0 on starting the script system.

If TRACE is executed without a parameter, then the trace state toggles: if trace was off, it gets set to level 1. If it was on at any level, it gets turned off.

The currently supported trace levels are:

- 0: Trace is off. Normal program execution
- 1: Show each line as it is about to be executed (just before execution)
- 2: Level 1 plus show each variable assignment as it happens
- >2: Treated as 2, at present.

PRINT {<expression>{, <more expressions>}}

PRINT is used to send data to the currently selected serial port (use the SETCOMPORT function to switch between use of the primary serial port and the SIO2 port). If PRINT is requested with no expressions, it sends a <CR><LF> (Carriage-return, line-feed) character pair.

```
PRINT "6*3=", 6*3
```

Would print out the text

```
6*3=18
```

It would NOT print out anything after the 18, so your next PRINT output would immediately follow the "8".

PRINTLINE {<expression>{, <more expressions>}}

PRINTLINE is used to send data to the currently selected serial port (use the SETCOMPORT function to switch between use of the primary serial port and the SIO2 port).

The list of text is always followed by a <CR><LF> (Carriage-return, line-feed) character pair.

```
PRINTLINE "6*3=", 6*3
```

Would print out the text

```
6*3=18
```

It would print out a <CR><LF> after the above text, so the next print would be on a new line.

Variable Assignment: <VAR>=<EXPRESSION>

A variable assignment consists of a variable reference, followed by the "=" character, followed by a valid expression. For example,

```
A=10           ' Sets integer variable A to a value of 10
A1$(3) = 4     ' Sets the fourth character in A1$ to a binary value of 4
A=REPORTEXTENDEDINPUTS ' Sets A to the value of the extended inputs register
```

An expression of the form "<VAR>=<EXPRESSION>" that is found elsewhere on a line is NOT an assignment, but rather is a comparison operation. I.e.,

```
A=10           ' This sets integer variable A to 10
B=A=10         ' This sets B to the logical result of comparing A to value 10
```

Special case for string variable assignment – string concatenation

When assigning a string variable, the assignment process adds a few more rules:

- The assignment may be to a comma separated list. The result will be the concatenation of all of the list elements into one string
- If a list element is a string or a string variable, the entire string is appended (be careful! Some combinations will wipe out all memory!)
- If a list element is numeric, the number gets converted to text and appended to the string
- If a list element is a single character from a string variable, then that single character is appended as a character.

For example, the sequence

B$="Bill"

C=14.52

A$="Hi There, ", B$, "! You owe $",C," on your account."

Would end up with A$ containing the same text that it would have from

A$="Hi There, Bill! You owe $14.52 on your account."

This effectively means that a string assignment is almost identical to a print statement, excluding issues cause by memory overwrites.

As a critical caveat, the only upper length on a string is the end of the fixed variable array. If you do an assignment in such a way that the end-of-string null character gets overridden while doing the action, you will get the rest of the variable space scrambled.

For example,

A$="01234567890"

Generates A$ as a 10 character string. If you then attempt to concatenate it to itself, it will fill all of memory… This is due to there being no 'scratch space': During a string concatenation event, the direct variable array is used.

A$=A$,A$

Will destroy all of your variables!

GOTO <Expression> transfers control to a new line

The program may be instructed to jump to any labeled line. The <expression> gets evaluated to an integer, which is treated as the line number to use as the target of the goto. Note that it is explicitly legal to do a "computed goto", wherein you calculate which line is to be accessed. For example,

```
10 GOTO 100           ‘ Just go to line 100
20 A=3
30 GOTO 100+A*10      ‘ Go to a calculated line relative to line 100
40 END
100 PRINT "At line 100"
110 PRINT "AT LINE 110"
120 PRINT "AT LINE 120"
130 PRINT "AT LINE 130"
```

The statement at line 10 would just transfer to the statement at line 100. The statement at line 30 would transfer to the statement at line 130, since A is 3 at the time that it is used (assuming that line 20 were the starting point).

GOSUB <Expression> allows a line to be called as a subroutine

GOSUB is identical to GOTO, except that the current program location is remembered by the scripting system. If a RETURN statement is later executed, then the program will resume at the line following that which held the GOSUB.

```
10 GOSUB 100          ‘ Call the routine at line 100
20 A=3
30 GOSUB 100+A*10     ‘ call to a calculated line relative to line 100
40 END
100 PRINT "At line 100"
110 PRINT "AT LINE 110"
120 PRINT "AT LINE 120"
130 PRINT "AT LINE 130"
140 RETURN
```

RETURN resumes execution on the statement following the last GOSUB

RETURN is used to return from a GOSUB action. See the above example…

IF <EXPRESSION> THEN <EXPRESSION-TRUE> {ELSE <EXPRESSION-FALSE>}

The IF…THEN…ELSE construct allows the program to go to target locations based on testing the value of the requested expression. The ELSE clause is optional, and may be used to specify the transfer target if the specified expression is false.

An <EXPRESSION> is considered to be true if it is non-0, false if it is 0.

The goto target expressions are treated identically to the expressions with the GOTO command: they may include calculations in order to generate the target line number.

```
10 A=3
20 IF A=4 THEN 100
30 IF A=0 THEN 200 ELSE 300+10*A
40 END
100 PRINT "AT LINE 100: <A> WAS 4"
110 END
200 PRINT "AT LINE 110: <A> WAS 0"
210 END
300 PRINT "THIS LINE WILL NOT BE EXECUTED"
310 PRINT "THIS LINE WOULD BE EXECUTED IF A WERE 1"
320 PRINT "THIS LINE WOULD BE EXECUTED IF A WERE 2"
320 PRINT "THIS LINE WILL BE EXECUTED SINCE A WAS 3"
330 END
```

FOR <Var Assignment> TO <expression> {STEP <expression 2>}

This starts up a complete looping system. The requested variable gets set as the active looping variable and is assigned its initial value.

Code will then execute normally until the associated NEXT is found, and then the code will update the variable by the requested STEP amount, and will compare the resulting variable to the saved value of the "TO" expression to determine if it can loop. If it can, it will execute a GOTO to the line following the related FOR command.

```
10 A$ = "This is a test"
20 FOR I = 0 TO 14
30 PRINT "A$(" , I , ")=" , A$ ( I ) , " "
40 J = A$ ( I )
50 PRINT "j=" , J
60 PRINT
70 NEXT
80 END
```

When run, it generates:

```
A$(0)=T j=84
A$(1)=h j=104
A$(2)=i j=105
A$(3)=s j=115
A$(4)=  j=32
A$(5)=i j=105
A$(6)=s j=115
A$(7)=  j=32
A$(8)=a j=97
A$(9)=  j=32
A$(10)=t j=116
A$(11)=e j=101
A$(12)=s j=115
A$(13)=t j=116
A$(14)= j=0
```

NEXT starts up the next cycle in a FOR…NEXT loop

NEXT is used as an adjunct to the FOR command (above). It actually causes the next value to be calculated for the loop control variable, and executes the GOTO to perform the next iteration if the resulting value permits it.

Poor Man's Parameter Passing – SetParameterList, ParameterCount and Parameter

As of version 5.42, the scripting system supports a very simplistic standardized parameter passing system, to simplify coding of the equivalent of function parameters for subroutines.

The "**SetParameterList**" function may be used to initialize an array of up to 10 generic parameters, which may then be used as value inputs by subroutines in your program until the next "SetParameterList" call is made.

The **ParameterCount** function reports the current count of defined parameters in that list.

The **Parameter**(I) function reports the current value for parameter number I (counting from 0 to ParameterCount-1).

For example,

```
10 SETPARAMETERLIST( 1 , 2 , 3 , "hi there!" )
20 PRINT "Parameter count = " , PARAMETERCOUNT , "\r\n"
30 FOR I = 0 TO PARAMETERCOUNT - 1
40 PRINT "Parameter " , I , " = " , PARAMETER( I ) , "\r\n"
50 NEXT
Program currently occupies 156 bytes.  There are 8036 bytes of program space left.
READY
run
Parameter count = 4
Parameter 0 = 1
Parameter 1 = 2
Parameter 2 = 3
Parameter 3 = hi there!
READY
```

Please be aware that string parameters (such as the "hi there" in the above example) are actually only saved as pointers to the string in the static parameters array. This means that if you use a variable as the source of a string, and you then modify that variable, the string contents will be changed for that parameter.

Special Serial Functions

The scripting system has several functions that aid in normal program operation, but are not really “motor control” actions. These functions control the serial I/O system (i.e., they select which serial port to use, and allow access to those ports). The PRINT statement makes use of the currently selected serial port to generate its output, and the remaining functions described here provide additional serial control.

The script system **always** monitors the primary serial input stream (port 0) for a CTRL-C character as being the next character to process. If it sees that character, it will abort any executing script. The ‘SetVerboseMode’ function (the ‘V’ command in the non-scripting system) also has a control bit that tells the low level serial buffering system to automatically purge any unprocessed serial characters when a Ctrl-C is seen, so that an incoming CTRL-C becomes the next character to be processed, period.

Similarly, if the serial system sees a Ctrl-C as the next character to be processed on actually extracting a character on the alternate serial port (port 1, SI2/SO2), the program will be stopped.

A=SETBOARDBAUDRATE(<expression>) defines the serial baud rate

SETBOARDBAUDRATE is used to set the baud rate for communication on both the primary and the secondary serial ports. <expression> is any valid baud rate from 2400 through 115200 baud.

Note: This is identical to the issuing the ‘b’ command in the non-scripting system.

A=SETCOMPORT(<expression1> {, <stop string>, {<skip string>}}) selects the com port to use and how to process the input text

SETCOMPORT is used to set which serial port is to be used for PRINT output and any serial input commands. The value of <expression1> controls the port: 0 means use the main I/O serial port (i.e., the USB connection), 1 means use the SI2/SO2 ttl-serial port if it has already been configured as a serial port.

If provided, the <stop string> text may be used to list characters that will stop INPUTLINE$ from reading text data (for example, “*” would make INPUTLINE$ stop on seeing an ‘*’ instead of a <CR> character). If the stop string is not provided or is empty, <CR> is used.

If provided, the <skip string> text may be used to list characters that are to be skipped on the input. For example, “ h” would skip all space and ‘h’ characters. If the skip string is not provided, then no characters are skipped.

Once SETCOMPORT has been executed, all serial I/O is redirected to the indicated port. If there is a program error or the program exits (via EXIT or END), the port gets reset to port 0, the standard serial port.

A=INPUTCHARACTER gets the next input character

The INPUTCHARACTER function waits for the next character to be sent via the current serial port, and returns that character as an ASCII character integer (value of 0 to 255).

As with INPUTLINE$, if the next character is a CTRL-C, the program gets aborted.

A=INPUTCHARACTERIFANY gets the next input character, if any

The INPUTCHARACTERIFANY function extracts the next character to be sent via the current serial port, and returns that character as an ASCII character integer (value of 0 to 255).

If there is no character currently available, then the function reports a value of -1. This allows your program to cycle, and to just “poll” the input as needed.

As with INPUTLINE$, if the next character is a CTRL-C, the program gets aborted.

A$=INPUTLINE$ inputs a line of text

The INPUTLINE$ function inputs a line of text from the currently selected port. If the target of the operation is an assignment to a string ("A$=INPUTLINE$"), then the entire line input is copied into that string. If the target requires a numeric value (such as assigning to a floating point variable, as in "A#=INPUTLINE$"), then the appropriate conversion is done. A single static buffer is used for the input, so the maximum length of text input using this method is 255 characters. Any extra characters until the termination character are dropped.

If a CTRL-C is sent via that serial port, the program gets immediately aborted.

As an example of 'talking' to the alternate serial port, we get:

```
1 ' Trivial Program to show connection via the alternate COM port
5 SETIOCONFIG( 32 ) ' This enables the second serial port
10 SETCOMPORT( 1 ) ' Select I/O via the alternate port
20 PRINT "Hi There!\r\nPlease enter some text:"
30 A$ = INPUTLINE$
40 PRINT "\r\nYou typed <" , A$ , ">\r\n"
50 SETCOMPORT( 0 )
60 PRINT "The other terminal typed <" , A$ ">\r\n"
```

Only use at most one function from the list of LEFT$, MID$, RIGHT$, READ$ and INPUTLINE$ in an expression! Since they all use same text buffer for their internal output result, a complex expression that uses more than one of those functions is likely to misbehave.

A=LASTINPUTCHARACTER gets the last character input by INPUTLINE$, INPUTCHARACTER or INPUTCHARACTERIFANY

The LASTINPUTCHARACTER function reports the last character input by the INPUTLINE$, INPUTCHARACTER or INPUTCHARACTERIFANY functions. This allows you to detect which character terminated a "line" on the INPUTLINE$ function. Once the program exits (i.e., once you are back into command line mode), LASTINPUTCHARACTER gets reset to 0.

A=PEEKCHARACTER snoops on the next input character on port 0, if any

The PEEKCHARACTER function extracts a copy of the next character to be sent via the serial port 0, and returns that character as an ASCII character integer (value of 0 to 255). It does NOT remove that character from the input queue; the next call to INPUTCHARACTER or INPUTCHARACTERIFANY will retrieve the same character.

Please note: PEEKCHARACTER only works on port 0. It ignores the current port assignment (SETCOMPORT).

If there is no character currently available, then the function reports a value of -1. This allows your program to cycle, and to just "poll" the input as needed.

As with INPUTLINE$, if the next character is a CTRL-C, the program gets aborted.

Generic Support Functions

The scripting system supports several generic functions of use in math and string processing.

A=ABS(<value>) reports the absolute value of the requested expression

ABS may be used to generate the absolute value of the enclosed expression. For example,

    A=ABS(32-74)

Would report 42 as the assignment result.

A=INSTR(<Haystack>, <Needle>) will attempt to find <Needle> in <Haystack>

The INSTR function requires that <Haystack> be a string. It will search for <Needle> in <Haystack>, reporting the character offset to the start of the requested text. It will report a "-1" if <needle> cannot be found or if <Haystack> is not a string.

If <Needle> is numeric, it gets converted to a character (after taking it modulus 256), and the first occurrence of the resulting character is reported.

For example,

    A=INSTR("Hi there", "he")

Will set A to 4. Similarly,

    A=INSTR("Hi There", 32)

Will set A to 2 (32 is the ASCII value of the space character).

A=LEN(<string>) reports the count of characters in a string

LEN reports the count of characters in a string, not including the trailing null.

Note that it will report 0 if used on non-text items

    A=LEN("George")

Will set A to 6.

A$=LEFT$(B$, count) reports the leftmost <count> characters in B$

LEFT$ reports the first <count> characters of B$ using a fixed internal temporary buffer (the same one used by INPUTLINE$, MID$ and RIGHT$). The maximum string length returned by LEFT$ is 255 characters, due to use of this fixed buffer.

If <count> is greater than the length of B$, all of B$ (up to the first 255 characters) will be reported.

Only use at most one function from the list of LEFT$, MID$, RIGHT$, READ$ and INPUTLINE$ in an expression! Since they all use same text buffer for their internal output result, a complex expression that uses more than one of those four functions is likely to misbehave.

A$=MID$(B$, offset, count) reports the middle <count> characters in B$ starting at the requested offset location

MID$ reports up to <count> characters of B$ starting at the requested offset using a fixed internal temporary buffer (the same one used by INPUTLINE$, LEFT$ and RIGHT$). The maximum string length returned by MID$ is 255 characters, due to use of this fixed buffer.

If <offset+count> is greater than the length of B$, all of B$ (up to the first 255 characters) from location offset on will be reported.

If offset is greater than the length of B$, then a string of 0 length will be reported.

Only use at most one function from the list of LEFT$, MID$, RIGHT$, READ$ and INPUTLINE$ in an expression! Since they all use same text buffer for their internal output result, a complex expression that uses more than one of those functions is likely to misbehave.

A$=RIGHT$(B$, count) reports the rightmost <count> characters in B$

RIGHT$ reports the last <count> characters of B$ using a fixed internal temporary buffer (the same one used by INPUTLINE$, LEFT$ and MID$). The maximum string length returned by RIGHT$ is 255 characters, due to use of this fixed buffer.

If <count> is greater than the length of B$, all of B$ (up to the first 255 characters) will be reported.

Only use at most one function from the list of LEFT$, MID$, RIGHT$, READ$ and INPUTLINE$ in an expression! Since they all use same text buffer for their internal output result, a complex expression that uses more than one of those functions is likely to misbehave.

A=SIN(<angle>) will return the sine of the requested angle

SIN reports the sine of the requested angle, with the angle expressed in degrees.

    A=SIN(45)

Will set A to (about) 0.7071….

A=COS(<angle>) will return the cosine of the requested angle

COS reports the cosine of the requested angle, with the angle expressed in degrees.

    A=COS(45)

Will set A to (about) 0.7071….

DATA <text> creates in-program data for the READ command and the READ$ function

DATA is used to save comma-separated data for use by the READ$ function. This allows your program to have a list of built-in values that can be used to fill in variables as needed

If a comma-separated element begins with a double-quote character ("), then all data until the matching quote (or until the end of the line) will be included as one data item. Additionally, while in the quote-processing mode, a "\" character may be used to 'escape' the next character's default interpretation, so that it will be included as part of the string. This is used to allow the '"' character to be included as part of the text.

For example,

```
10 DATA Hi, there folks!, "we are ready, willing and able", 35.7
```

would have data items:

```
Hi
There folks!
we are ready, willing and able
35.7
```

Available for reading via the "READ" command and the "READ$" function.

READ <variable list> reads data in DATA statements into the selected variables in the <variable list>

READ is used to read data from the list of DATA statements in the program into a selected list of variables. The variable list is a comma separated list of variable names, which may include indexed variables (such as A(10) or A(c*3)).

For example,

```
10 DATA Hi, there folks!, "we are ready, willing and able", 35.7
20 READ A$, B$, C$, F(7)
```

would have data items (after a run) of:

```
A$="Hi"
B$="there folks!"
C$="we are ready, willing and able"
F(7)=35.7
```

Note that the execution of the READ command is strictly left-to-right in terms of processing any array elements, so it is expressly legal to do the following:

```
10 DATA 6,2
20 READ I,A(I)
```

This would result in the values of

```
I=6
A(6)=2
```

RESTORE <optional line number> resets the DATA pointer to the first program line that starts with DATA at or after the indicated line number

RESTORE is used to set the internal pointer to the requested line of data. If no line number is provided, it will restore the data pointer to the first data line in the program, if any.

RESTORE

Resets to the first DATA line, while

RESTORE 230

Resets to the first data line at or after line 230.

A=READ$ reads the next data element

The READ$ function reports the next data element from the internal data list as a string. You can assign the result to any type of variable, since the interpreter will automatically convert types as needed.

For example,

```
10 DATA 0, 4, "Hi"
20 A=READ$
30 B=READ$
40 C$=READ$
```

Would assign:

```
A=0
B=4
C$="Hi"
```

Only use at most one function from the list of LEFT$, MID$, RIGHT$, READ$ and INPUTLINE$ in an expression! Since they all use same text buffer for their internal output result, a complex expression that uses more than one of those functions is likely to misbehave.

Starting up a script on board reset

The firmware supports running a saved script as part of the board reset/startup process. The script is run after the copyright message is sent, before the code enters its monitoring loop for incoming command characters.

To do this, do the following:

1. Create your program. This must be a program with line numbers, as opposed to just single-line commands. Only line-number based statements are saved.
2. Save the program by requesting "SAVE". This will save the program into flash memory
3. Exit the simple "2[" mode (using the "EXIT" command), and instead create a startup-"run" command by sending the text "1[[run"
4. Memorize the startup command with the "-123456789e" command

For example, you could end up with the following 'conversation' via a terminal emulator:

```
*2[
10 PRINT "IT WORKED!\r\n"
EXIT
*1[
[RUN
*-123456789e
User Flash write request complete!
*!
M2,-12
SD6DXNCRouter 5.27 December 29, 2015
Copyright 2014, 2015 by Peter Norberg Consulting, Inc.  All Rights Reserved.
*It worked!
*
```

The key on this is the "1[" command, which allows you to specify a set of commands to execute on a reset or restart action. As of firmware version 5.27, if the startup process sees a "[" character in that sequence, it sends all of the text following that "[" to the scripting command interpreter after the rest of the startup process has completed (after the copyright text has been printed). "run" is the command that will run the current script from the beginning. You could instead do a "goto" command in order to start at a given line number in your program.

WARNING! It is quite possible to make a board be incapable of being reprogrammed when a startup script is enabled.

If you enable a startup script, the FirmwareUpdater is not likely to be able to correctly synchronize with the board. You will need to erase your flash memory via the

-987654321e

command before the FirmwareUpdater will work.

Board Functions

The remaining functions are direct connections to the standard serial command system (internally, they simply directly emulate the serial system commands). They are listed earlier in this manual as part of each command (for example, the "j" command lists the SETMICROSTEPSIZE function as its matching command), and are repeated here for completeness.

ASYNCHGOTOLOCATION(<idMotor>, <Location>)

ASynchGoToLocation tells the indicated set of motors (as defined by the bit-mapped integer idMotor; see the 'm' command) to go to the requested location. The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

    ASynchGoToLocation(2,3000)

Would tell motor X to go to location 3000.

Critical Change - As of version 5.47, this function respects the current setting of the "SetAddressMode" state. If the current address mode is relative, this does a relative seek. If the mode is absolute, this does an absolute goto. All versions before 5.47 treat all AsynchGoToLocation requests as absolute.

ASYNCHGOTOMARKEDLOCATION(<idMotor>)

ASynchGoToMarkedLocation tells the indicated set of motors (as defined by the bit-mapped integer idMotor; see the 'm' command) to go to their previously marked location (see ASynchMarkCurrentLocation). The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

    ASynchGoToMarkedLocation(4)

Would tell motor Y to go to its previously marked location.

ASYNCHMARKCURRENTLOCATION(<idMotor>)

ASynchMarkCurrentLocation saves the current location for the indicated motor(s) (as defined by the bit-mapped integer idMotor; see the 'm' command) as the target to use on a subsequent ASynchGoToMarkedLocation call. The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

    ASynchMarkCurrentLocation(63)

Would save all of the current locations as future ASynchGoToMarkedLocation targets.

ASYNCHSETLOCATION(<idMotor>, <Location>)

ASynchSetLocation tells the indicated set of motors (as defined by the bit-mapped integer idMotor; see the 'm' command) that their current location is defined as <Location>. It also instantly aborts any motion on the selected motors. The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

    ASynchSetLocation(2,3000)

Would tell motor X that its current location is defined as 3000, and it would halt any motion on motor X.

ASYNCHSETRAMPRATE(<idMotor>, <RampRate>)

ASynchSetRampRate sets the ramp rate for the indicated motor(s) (as defined by the bit-mapped integer idMotor; see the 'm' command). The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

    ASynchSetRampRate(15, 60000)

Would set the ramp rates for motors W, X, Y and Z to 60,000 microsteps/second/second.

ASYNCHSETRUNRATE(<idMotor>, <RunRate>)

ASynchSetRunRate sets the target run rate for the indicated motor(s) (as defined by the bit-mapped integer idMotor; see the ‘m’ command). The motors only respond to the request if they have been configured as asynchronous motors via the ‘T’ command, SetLimitSwitchEnables.

```
ASynchSetRunRate(7, 10000)
```

Would set the run rates for motors W, X, and Y to 10,000 microsteps/second.

ASYNCHSETSTOPOKRATE(<idMotor>, <StopRate>)

ASynchSetStopOKRate sets the start/stop rate for the indicated motor(s) (as defined by the bit-mapped integer idMotor; see the ‘m’ command). The motors only respond to the request if they have been configured as asynchronous motors via the ‘T’ command, SetLimitSwitchEnables.

```
ASynchSetStopOKRate(7, 1000)
```

Would set the start/stop rates for motors W, X, and Y to 1,000 microsteps/second.

ASYNCHSLEW(<idMotor>, <Direction>)

ASynchSlew starts the indicated motor(s) (as defined by the bit-mapped integer idMotor; see the 'm' command) spinning in the indicated direction. The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

<Direction> contains the request for the direction of spin: >=0 means spin in the positive direction, <0 means spin in the negative direction

```
ASynchSlew(16, -1)
```

Would cause the A motor to spin in the negative direction.

ASYNCHSTOPMOTOR(<idMotor>)

ASynchStopMotor stops motion on the indicated motor(s) (as defined by the bit-mapped integer idMotor; see the 'm' command). The motors only respond to the request if they have been configured as asynchronous motors via the 'T' command, SetLimitSwitchEnables.

```
ASynchStopMotor(32)
```

Would stop the B motor from spinning.

A=DIAGMAXINTTIME

DiagMaxIntTime reports a diagnostic count of the maximum clock ticks in the interrupt processing routine since the last time that this function was called. This is a developmental diagnostic function, and is not likely to be needed in "real" code.

DRAWARC(<Radius>, <BeginAngle>, <EndAngle>, <StepCount>)

DrawArc draws a complete arc of the indicated radius, using the last two motors accessed via the SetMotorLocation… commands. The angles are in degrees (and are processed as floating point). The <StepCount> states how many vectors to draw if positive, or the desired length of each vector if negative. A step count of 0 just goes to the starting location for the arc relative to the current location.

```
DrawArc(1000, 0, 360, 100)
```

Would thus draw a full circle (on the last two motors) of radius 1000 and composed of 100 short vectors.

DRAWARCWITHRATE(<Radius>, <BeginAngle>, <EndAngle>, <StepCount>, <Rate>)

DrawArcWithRate is identical to DrawArc, with the addition of allowing you to specify the desired target rate (as opposed to using the current rate).

```
DrawArcWithRate(1000, 0, 360, 100, 2000)
```

Would thus draw a full circle (on the last two motors) of radius 1000 and composed of 100 short vectors at a rate of 2000 microsteps/second.

DRAWSPIRAL(<BeginRadius>, <EndRadius>, <BeginAngle>, <EndAngle>, <StepCount>)

DrawSpiral draws a spiral starting at radius <BeginRadius> and ending at radius <EndRadius>, starting at angle <BeginAngle> and ending at angle <EndAngle>. It will draw it in <StepCount> vectors if StepCount is positive, or it will draw it in vector lengths that start at the requested length if StepCount is negative.

```
DrawSpiral(500, 1000, 0, 360, 100)
```

Would thus draw a full spiral (on the last two motors) of radius starting at 500 and increasing to 1000, composed of 100 short vectors.

DRAWSPIRALWITHRATE(<BeginRadius>, <EndRadius>, <BeginAngle>, <EndAngle>, <StepCount>, <Rate>)

DrawSpiralWithRate is fully identical to DrawSpiral, with the addition of allowing you to specify the desired target rate (as opposed to using the current rate).

    DrawSpiralwithRate(500, 1000, 0, 360, 100, 2000)

Would thus draw a full spiral (on the last two motors) of radius starting at 500 and increasing to 1000, composed of 100 short vectors, at a rate of 2000 microsteps/second.

A=EEPROMREADDATA(<idWhichElement>)

EEPromReadData reads the selected EEProm element as defined by idWhichElement. The most common values for <idWhichElement> to use are:

0-31: User EEProm elements for any use you need

32: Board type

33: Board serial number within run

34: Manufacturing run number (encoded)

35: Manufacturing reference number

36: Special order flags

37 to 63: Reserved for future use

For example,

    A=EEPromReadData(33)

Will report the board serial number within the encoded manufacturing run number.

EEPROMRESET erases the EEProm data

EEPromReset erases all user data from the EEProm system. This deletes any saved script, any saved board settings, and the 32 elements of EEProm user data.

    EEPROMRESET

This restores the board to is factory defaults on the next board reset.

EEPROMSAVECURRENTSETTINGS

EEPromSaveCurrentSettings saves all of the memorizable parameters (such as target rates, motor currents, I/O configurations and so on) into the EEProm system. On the next board reset, those memorized settings will be applied.

    EEPromSaveCurrentSettings

Performs the memorize action.

EEPROMSAVEUSERSETTINGS

EEPromSaveUserSettings saves just the user data (the 32 element user data array and the user startup command) if the EEProm contents are currently valid. If they are not currently valid, then EEPromSaveUserSettings acts identically to EEPromSaveCurrentSettings, and saves everything that is can be saved.

    EEPromSaveUserSettings

Performs the memorize action.

EEPROMWRITEDATA(<idWhichEntry>, <value>)

EEPromWriteData writes the indicated user-accessable entry with the requested value. <idWhichEntry> must be from 0 to 31, and <value> is treated as a 32 bit integer.

    EEPromWriteData(4, 231)

Would write user entry 4 with a value of 231.

GOTOCURRENTREQUESTEDLOCATION

GoToCurrentRequestedLocation tells the vector system to queue a request to go to the currently requested location (as specified by the various SetRequestedLocation… functions). The function returns the count of queue elements that are still available after the request.

A=GoToCurrentRequestedLocation

Would start motion to the currently requested location, and would return a '97' if there are now 3 elements in the queue.

GOTOCURRENTREQUESTEDLOCATIONWITHVECTORLENGTH(<lLength>)

GoToCurrentRequestedLocationWithVectorLength is identical to GoToCurrentRequestedLocation, with the additional feature of specifying the target length for the entire vector (used for rate scaling). It tells the vector system to queue a request to go to the currently requested location (as specified by the various SetRequestedLocation… functions). The function returns the count of queue elements that are still available after the request.

A=GoToCurrentRequestedLocationWithVectorLength(1000)

Would start motion to the currently requested location and scaling the rates based on a vector length of 1000, and would return a '97' if there are now 3 elements in the queue.

GOTOLOCATION(<W>, <X>, <Y>, <Z>, <A>, <B>)

GoToLocation queues a request to go to the requested location. The function returns the count of queue elements that are still available after the request.

A=GoToLocation(10, 20, 30, 40, 50, 60)

Would request that the <W,X,Y,Z,A,B> motors move to location <10,20,30,40,50,60> as the next vector point. The function returns the available queue space after the call.

GOTOLOCATIONWITHVECTORLENGTH(<W>, <X>, <Y>, <Z>, <A>, <B>, <lLength>)

GoToLocationWithVectorLength is identical to GoToLocation, with the addition of allowing specification of the nominal vector length to use in rate calculations.

A=GoToLocationWithVectorLength(10, 20, 30, 40, 50, 60, 300)

Would request that the <W,X,Y,Z,A,B> motors move to location <10,20,30,40,50,60> as the next vector point, with rates scaled assuming that the total vector length is 300. The function returns the available queue space after the call.

GOTOLOCATIONWITHVECTORLENGTHANDRATE(<Rate>, <W>, <X>, <Y>, <Z>, <A>, <B>, <lLength>)

GoToLocationWithVectorLengthAndRate is identical to GoToLocationWithVectorLength, with the addition of allowing specification of the target rate for the vector.

A=GoToLocationWithVectorLengthAndRate(30000, 10, 20, 30, 40, 50, 60, 300)

Would request that the <W,X,Y,Z,A,B> motors move to location <10,20,30,40,50,60> as the next vector point, with a nominal target rate of 300 and with that rate scaled assuming that the total vector length is 300. The function returns the available queue space after the call.

A=IDLEWAIT(<idMotor>, <MaxWait>)

IdleWait waits for the select motors to be sufficiently idle to allow queueing of a new motion command. The idMotor selects the motors to be tested (the queue is always tested), and the MaxWait parameter describes the maximum time (in milliseconds) that the wait will take. Note that if a character is received, the idle wait will abort.

The function returns the type code for the last wait event:

0: Idle

1: Paused

2: Stopped

3: Programmed Delay

4: Drawing an arc

5: Any other item pending in the Queue

6: Any of the selected motors still being active (if asynchronous)

If the current wait type is 4 or less, IdleWait returns immediately.

A=IdleWait(3, 1000)

This will wait up to 1000 milliseconds (1 second), or until the wait test is satisfied on motors W and X.

A=IDLEWAITFORFULLYSTOPPED(<idMotor>)

IdleWaitForFullyStopped waits until all selected motors are completely idle and the queue is empty, or until there is a character received or the system is paused.

As with IdleWait, the function returns the current status of the system (see IdleWait for the values returned).

A=IdleWaitForFullyStopped(3)

This will wait until the motion queue is empty (or blocked from becoming empty) and for motors W and X to become idle.

A=LASTQUEUECOUNT reports available queue space

LastQueueCount reports an instant snapshot of the currently available space left in the queue.

WARNING! If you are drawing an arc, you may see a non-0 LastQueueCount while the arc is still active. You must wait until the LastWaitType is not in Arc mode to perform a new queued action (such as queing another vector).

A=LastQueueCount

Reports the current count of queue elements that are available.

A=LASTWAITTYPE reports the current queue status

LastWaitType reports the current instant status of the queue. The values returned are:

0: Idle

1: Paused

2: Stopped

3: Programmed Delay

4: Drawing an arc

5: Any other item pending in the Queue

6: Any of the selected motors still being active (if asynchronous)

For example,

A=LastWaitType

Would return a 0 if the queue is idle and the currently selected motors are idle.

A=MAXRATE reports the maximum rate supported by the current configuration

MaxRate reports the maximum possible run rate as supported by the current firmware.

A=MAXRATE

Would return 250000 for the BC6D25 board, and 500000 for the BC6D20 and SD6DX boards.

PAUSEMOTORS pauses the vector motion

PauseMotors pauses any vector motion that is underway (the LastWaitType will change to 1 once the pause has completed).

Motors may be restarted using the RequestMotorRestart function, or by issuing the STOPALLMOTORS or STOPVECTORMOTORS function (which flushes the queue and removes the pause flag).

```
PauseMotors
```

Starts the motors slowing down for the pause action.

PAUSEMOTORSWITHTTLRESTART(<idRestartOptions>, <idPort>)

PauseMotorsWithTTLRestart issues the pause request that may be resumed with a TTL line.

The <idRestartOptions> tell the firmware how to use a TTL line to restart, while the <idPort> describes which TTL port to monitor for the restart request.

idRestartOption	Use
0-1:	No TTL restart ('r' only)
2:	Restart when selected input low
3:	Restart when selected input high
4:	Restart when selected input edge goes low
5:	Restart when selected input edge goes high

idPort	Use
0	RDY
1	NXT
2	SI2
3	SO2
4	IO2
5	W-
6	W
7	X-
8	X+
9	Y-
10	Y+
11	Z-
12	Z+
13	A-
14	A+
15	B-
16	B+

For example,

```
PauseMotorsWithTTLRestart(3, 9)
```

Would pause the motors until the Y- input becomes high.

PAUSENOW(<PauseTime>) waits a requested amount of time

PauseNow is a programmatic delay for an indicated number of milliseconds. The firmware will wait for the requested time before processing the next statement.

WARNING! The serial input system is monitored during this pause. If an input character is available in the queue, then the pause will be abandoned.

For example,

```
PauseNow(3500)
```

Would delay 3.5 seconds.

A=QUEUEPAUSE(<PauseTime>) causes the vector system to delay PauseTime milliseconds

The QueuePause command queues a request to pause the requested number of milliseconds as part of the vector sequence. Once that part of the queue is reached, the vector system will pause the indicated number of milliseconds before resuming action.

WARNING! It is important that you set up your queue such that all vector motion is idle by the time that a QUEUEPAUSE event is seen. The pause is "instant", so any motors that are in motion will experience an "instant stop" when the pause starts. This is likely to result in missed steps during the stop. Similarly, when the pause completes, motion will resume AT THE SAME RATE THAT IT HAD WHEN IT STOPPED. This is likely to cause still more missed steps…

For example,

```
A=QueuePause(25)
```

Would insert a 25 millisecond pause in the next location in the motion queue.

It reports the current number of queue elements left after having queued the pause request.

A=QUEUEPAUSEMOTORSWITHTTLRESTART(<idRestartOptions>, <idPort>) queues a request to pause the motors with a TTL restart event being needed to start them back up.

QueuePauseMotorsWithTTLRestart is identical to PauseMotorsWithTTLRestart, except that the request is queued. It gets acted upon when that point is reached in the process queue.

The <idRestartOptions> tell the firmware how to use a TTL line to restart, while the <idPort> describes which TTL port to monitor for the restart request.

idRestartOption	Use
0-1:	No TTL restart ('r' only)
2:	Restart when selected input low
3:	Restart when selected input high
4:	Restart when selected input edge goes low
5:	Restart when selected input edge goes high

idPort	Use
0	RDY
1	NXT
2	SI2
3	SO2
4	IO2
5	W-
6	W
7	X-
8	X+
9	Y-
10	Y+
11	Z-
12	Z+
13	A-
14	A+
15	B-
16	B+

For example,

QueuePauseMotorsWithTTLRestart(3, 9)

Would queue a pause for the motors until the Y- input becomes high.

It reports the current number of queue elements left after having queued the pause request.

A=QUEUERATESET(<Rate>) Queues a an 'instant rate set' request

QueueRateSet inserts a new instant rate request into the next queue location.

It reports the current number of queue elements left after having queued the rate request.

An "instant rate" set is only valid if the current mode of the vector system is pure-user controlled rates (non-trapezoidal, obtained by requesting a SetRate using a negative value); otherwise, it is ignored.

A=QueueRateSet(4000)

Would force the rate to be at 4000 at that point in the queue, as an instant jump. It would report the number of queue elements left after having inserted the request in the queue.

A=QUEUESETIOANDSLEWPORTS(<IOBits>, <SlewBits>) queues a request to set the various I/O ports

QueueSetIOAndSlewPorts queues a request to set ALL programmable output bits to the desired values. <IOBits> defines the extended IO lines, while <SlewBits> defines the SLEW lines.

IOBits is formatted as:

Bit	**Value**	**Use**
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2

SlewBits is formatted as:

Bit	**Value**	**Use**
0	+1	W-
1	+2	W+
2	+4	X-
3	+8	X+
4	+16	Y-
5	+32	Y+
6	+64	Z-
7	+128	Z+
8	+256	A-
9	+512	A+
10	+1024	B-
11	+2048	B+
12	+4096	WO1
13	+8192	WO2
14	+16384	XO1
15	+32768	XO2
16	+65536	YO1
17	+131072	YO2
18	+262144	ZO1
19	+524288	ZO2
20	+1048576	AO1
21	+2097152	AO2
22	+4194304	B01
23	+8388608	B02

For example,

    A=QueueSetIOAndSlewPorts(1, 4096)

Would queue a request to set RDY and W01 high, and all other bits low. It would report the queue space available after the request has been queued.

A=QUEUESETIOANDSLEWPORTSHIGH(<IOBits>, <SlewBits>) queues a request to set the various I/O ports high

QueueSetIOAndSlewPortsHigh queues a request to set matching bits high when that point of the queue is reached. The format for IOBits and SlewBits matches that of QueueSetIOAndSlewPorts.

For example,

A=QueueSetIOAndSlewPortsHigh(1, 4096)

Would queue a request to set RDY and W01 high, leave all of the other bits unchanged. It would report the queue space available after the request has been queued.

A=QUEUESETIOANDSLEWPORTSLOW(<IOBits>, <SlewBits>) queues a request to set the various I/O ports low

QueueSetIOAndSlewPortslow queues a request to set matching bits low when that point of the queue is reached. The format for IOBits and SlewBits matches that of QueueSetIOAndSlewPorts.

For example,

A=QueueSetIOAndSlewPortsLow(1, 4096)

Would queue a request to set RDY and W01 low, leave all of the other bits unchanged. It would report the queue space available after the request has been queued.

A=QUEUESETIOPORTS(<AllIOBits>) queues a request to set all of the IO bits to the requested values

QueueSetIOAndSlewPorts queues a request to set ALL programmable output bits to the desired values. <AllIOBits> defines all of the TTL lines.

AllIOBits is formatted as:

Bit	Value	Use
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	B01
28	+268435456	B02

For example,

```
A=QueueSetIOPorts(131073)
```

Would queue a request to set RDY and W01 high, and all other bits low. It would report the queue space available after the request has been queued.

A=QUEUESETIOPORTSHIGH(<AllIOBits>) queues a request to set all of the IO bits that are set in the request high

QueueSetIOPortsHigh queues a request to set matching bits high when that point of the queue is reached. The format for AllIOBits matches that of QueueSetIOPorts.

For example,

A=QueueSetIOPortsHigh(131073)

Would queue a request to set RDY and W01 high, leave all of the other bits unchanged. It would report the queue space available after the request has been queued.

A=QUEUESETIOPORTSLOW(<AllIOBits>) queues a request to set all of the IO bits that are set in the request low

QueueSetIOPortslow queues a request to set matching bits low when that point of the queue is reached. The format for AllIOBits matches that of QueueSetIOPorts.

For example,

A=QueueSetIOPortsLow(131073)

Would queue a request to set RDY and W01 low, leave all of the other bits unchanged. It would report the queue space available after the request has been queued.

A=REPORTATODATMAXPOTRATE(<idMotor>)

ReportAToDAtMaxPotRate reports the value for the A/D reading that is associated with the maximum potentiometer rate for the selected motor.

A=ReportAToDAtMaxPotRate(4)

Reports the A/D reading associated with the Y motor maximum potentiometer rate setting.

A=REPORTATODATMINPOTRATE(<idMotor>)

ReportAToDAtMinPotRate reports the value for the A/D reading that is associated with the minimum potentiometer rate for the selected motor.

A=ReportAToDAtMinPotRate(4)

Reports the A/D reading associated with the Y motor minimum potentiometer rate setting.

A#=REPORTATODVALUE(<idChannel>)

ReportAToDValue reports the selected A/D reading.

The <idChannel> tells the code which A/D value is to be reported.

idChannel	**Meaning**
0	RDY
1	NXT
2	IO2
3	<reserved>
4	Chip temperature, as a raw reading
5	Chip temperature, in Celcius*10
6	Chip temperature, in Fahrenheit*10
7	Chip temperature, in Kelvin*10
8	RDY as a true voltage (0 to 3.3 volts)
9	NXT as a true voltage (0 to 3.3 volts)
10	IO2 as a true voltage (0 to 3.3 volts)
11	<reserved>
12	Chip temperature, as a raw reading in volts
13	Chip temperature in Celcius
14	Chip temperature in Fahrenheit
15	Chip temperature in Kelvin

For example,

```
10 FOR I=0 to 15
20 PRINT "Channel ", I, " = ", ReportAToDValue(I), "\r\n"
30 NEXT

run
Channel 0 = 4092.792
Channel 1 = 4077.958
Channel 2 = 4086.083
Channel 3 = 0
Channel 4 = 2138.625
Channel 5 = 183.168
Channel 6 = 649.2493
Channel 7 = 2914.67
Channel 8 = 3.288116
Channel 9 = 3.288855
Channel 10 = 3.285666
Channel 11 = 0
Channel 12 = 1.722107
Channel 13 = 18.32688
Channel 14 = 65.01557
Channel 15 = 291.467
READY
```

A$=REPORTCOPYRIGHTTEXT(<idWhich>) reports a portion of the copyright text statement

ReportCopyrightText reports a string containing a part of the the copyright statement. The portions reported are controlled by idWhich:

0:	Firmware name
1:	Extension to firmware name (special order)
2:	Firmware revision number
3:	Firmware date
4:	Copyright text

For example,

```
list
10 FOR I = 0 TO 4
20 PRINT "ReportCopyrightText(" , I , ")=<" , REPORTCOPYRIGHTTEXT( I ) , ">\r\n"
30 NEXT
Program currently occupies 95 bytes.  There are 8097 bytes of program space left.
READY
run
ReportCopyrightText(0)=<SD6DXNCRouter>
ReportCopyrightText(1)=<>
ReportCopyrightText(2)=<5.24>
ReportCopyrightText(3)=<December 24, 2015>
ReportCopyrightText(4)=<Copyright 2014, 2015 by Peter Norberg Consulting, Inc.
All Rights Reserved.>
READY
```

A=REPORTCURRENTENCODERLOCATION(<idWhich>) reports the current value for the selected encoder

ReportCurrentEncoderLocation reports the current location for the selected encoder.

idWhich is 1 for encoder 0, 2 for encoder 1.

For example,

A=ReportEncoderLocation(1)

Would report the current Encoder 0 reading.

A=REPORTCURRENTLOCATION(<idWhichMotor>) reports the current location for the selected motor

ReportCurrentLocation reports the instantaneous current location for the selected motor.

idWhichMotor is mapped as:

idWhichMotor	Motor
1	W
2	X
4	Y
8	Z
16	A
32	B

For example,

A=ReportCurrentLocation(4)

Would report the current location for motor Y.

A=REPORTCURRENTSTEPACTION(<idWhichMotor>) reports the current state for the selected motor

ReportCurrentStepAction reports the motion status of the indicated motor (idWhichMotor is encoded as shown in ReportCurrentLocation). The values reported are:

0: Idle; all motion complete
1: motor in motion; either ramping up or at target speed
2: ramping down; either due to speed variation request or due to a stop request.
3: Slewing ("+$S")
4: Ramping down on a quick stop action
5: reversing direction of a slew
6: New goto detected: doing extended motor actions to make the transition as smooth as possible
7: Vector motion is active on this motor
8: If motor is in vector mode, this value will normally appear if this motor has no motion during this vector

For example,

```
A=ReportCurrentStepAction(2)
```

Would report a 0 if motor X was fully idle.

A=REPORTEXTENDEDINPUTS reports the current value for the extended inputs

ReportExtendedInputs provides a snapshot reading of the 5 extended I/O lines.

The report value is bit mapped as follows:

Bit	Value	Description
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2

A=REPORTFIRMWAREVERSION reports the current firmware version as an integer

ReportFirmwareVersion reports the current firmware revision level as an integer (this is the standard firmware version multiplied by 100).

For example,

```
A=ReportFirmwareVersion
```

Would return a value of 524 to express firmware version 5.24.

A=REPORTIOCONFIG reports the current IOConfig setting

ReportIOConfig returns the current IOConfig setting (see SetIOConfig for the mappings).

A=REPORTIOPORTDIRECTIONS reports the current I/O port directions

ReportIOPortDirections returns the current I/O port direction settings (see SetIOPortDirections for the mappings).

A=REPORTLATCHEDDATA reports the current snapshot latched data

ReportLatchedData excutes the 'L' command, to return a report of the currently latched events. It then clears those latched events, in preparation for detecting new events.

The latched events reported are as follows:

Bit	***Value***	***Description***
0	+1	W- limit reached during a W- step action
1	+2	W+ limit reached during a W+ step action
2	+4	X- limit reached during a X- step action
3	+8	X+ limit reached during a X+ step action
4	+16	Y- limit reached during a Y- step action
5	+32	Y+ limit reached during a Y+ step action
6	+64	Z- limit reached during a Z- step action
7	+128	Z+ limit reached during a Z+ step action
8	+256	A- limit reached during an A- step action
9	+512	A+ limit reached during an A+ step action
10	+1024	B- limit reached during a B- step action
11	+2048	B+ limit reached during a B+ step action
12-15		Reserved for two more motors
16	+65536	System power-on or reset ("!") has occurred
17	+131072	Motion is blocked via TTL input (NXT input is at its 'block motion' level, if so configured)
18	+262144	If set, the precision 50 PPM clock is being used as the clock source. If clear, the internal 3% oscillator is being used as the clock source.
19-23		Reserved: currently all 0
24	+16777216	BC6D25: W motor driver is reporting a fault
25	+33554432	BC6D25: X motor driver is reporting a fault
26	+67108864	BC6D25: Y motor driver is reporting a fault
27	+134217728	BC6D25: Z motor driver is reporting a fault
28	+268435456	BC6D25: A motor driver is reporting a fault
29	+536870912	BC6D25: B motor driver is reporting a fault

For example, after initial power on,

A=ReportLatchedData

Could report a value of 65536.

A=REPORTLIMITSWITCHENABLES(<idWhichMotor>) reports the current limit switch enable settings

ReportLimitSwitchEnables returns the current limit switch settings for the given motor, as requested by the SetLimitSwitchEnables function. See SetLimitSwitchEnables for documentation of the format of the data returned.

A=REPORTLIMITSWITCHES reports the current limit switch readings

ReportLimitSwitches reports the current values for all of the limit switch inputs.

The report value is bit mapped as follows:

Bit	**Value**	**Description**
0	+1	LW- (W- limit)
1	+2	LW+ (W+ limit)
2	+4	LX- (X- limit)
3	+8	LX+ (X- limit)
4	+16	LY- (Y- limit)
5	+32	LY+ (Y+ limit)

6	+64	LZ- (Z- limit)
7	+128	LZ+ (Z- limit)
8	+256	LA- (A- limit)
9	+512	LA+ (A+ limit)
10	+1024	LB- (B- limit)
11	+2048	LB+ (B+ limit)

A=REPORTMICROSTEPSIZE(<idMotor>)

This reports the microstep size associated with the requested motor. On the BC6D20, the units are 1/16th of a step, so this would range from 1 to 16 (as perfect powers of 2). On the BC6D25 this is reported in units of 1/32nd of a step, so it would range from 1 to 32 (as perfect powers of 2). On the SD6DX this will always report a value of 1.

For example,

    A=ReportMicrostepSize(2)

Would report the microstep size associated with the X motor.

A=REPORTMOTORBACKLASH(<idMotor>)

ReportMotorBacklash reports the backlash setting for the indicated motor.

For example,

    A=ReportMotorBacklash(16)

Would report the backlash setting for the A motor.

A=REPORTMOTORSWITCHES(<idMotor>)

ReportMotorSwitches reports the 6 TTL lines associated with the requested motor.

Bit	Value	Description
0	+1	LM- (motor '-' limit)
1	+2	LM+ (motor '+' limit)
2	+4	M- (motor '-' slew)
3	+8	M+ (motor '+' slew)
4	+16	O1 (motor O1 output, SD6DX)
5	+32	O2 (motor O2 output, SD6DX)

For example,

    A=ReportMotorSwitches(1)

Would report the TTL lines associated with the W motor.

A=REPORTPRECISIONTIMER provides a snapshot of the free-running precision timer

ReportPrecisionTimer reports a 32 bit high speed timer that is always running. The timer is operated at 80 megahertz, so it wraps every 50 seconds or so.

This call should only be used to calculate short duration events, by first getting a base reading and then doing deltas.

For example,

```
A=ReportPrecisionTimer

<do something you want timed>

B=ReportPrecisionTimer – A
```

B will be the number of 1/80th of a microsecond clocks that occurred between the two readings.

A=REPORTPULSEWIDTH reports the currently requested step pulse width

ReportPulseWidth reports the pulse width (in microseconds) that is currently enabled on the SD6DX board. On the BC6D20 and BC6D25 it will always report a value of 1.

For example,

```
A=ReportPulseWidth
```

Will report a value of 2 if pulse widths are currently 2 microseconds.

A=ReportQueueElement(<idWhich>) reports one of the queue snapshot items

ReportQueueElement gives array-like access to the queue element captured by the last SetReportQueueElementSnapshot command. The parameter passed defines which portion of the resulting data is to be returned:

- 0: Report the queue element ID as requested in the SetReportQueueElementSnapshot command
- 1: Report the execution ID associated with the element
- 2: Report the command associated with the element
- 3: report the float parameter associated with the element
- 4-10: report one of the long parameters (0 to 6: subtract 4 from idWhich to identify the parameter) associated with the element

A=ReportQueueElementID reports the queue element snapshot ID

ReportQueueElementID reports the queue element ID as requested in the SetReportQueueElementSnapshot command

A=ReportQueueElementCurrentExecutionID reports the queue element snapshot execution ID

ReportQueueElementCurrentExecutionID reports the execution ID associated with the element captured by the prior SetReportQueueElementSnapshot command.

A=ReportQueueElementCommand reports the queue element snapshot command

ReportQueueElementCommand reports the command associated with the element captured by the prior SetReportQueueElementSnapshot command.

A#=ReportQueueElementFloatParameter reports the queue element snapshot float parameter

ReportQueueElementFloatParameter reports the floating point parameter associated with the element captured by the prior SetReportQueueElementSnapshot command.

A=ReportQueueElementLongParameter(idWhich) reports the selected queue element snapshot long parameter

ReportQueueElementLongParameter reports the requested long parameter associated with the element captured by the prior SetReportQueueElementSnapshot command.

idWhich identifies the parameter, and may be from 0 to 6.

A=REPORTSLEWSWITCHES reports the current slew switches

ReportSlewSwitches is used to provide a snapshot of the SLEW and IO port input values. The report value is bit mapped as follows:

Bit	Value	Description
0	+1	W- (W- slew)
1	+2	W+ (W+ slew)
2	+4	X- (X- slew)
3	+8	X+ (X- slew)
4	+16	Y- (Y- slew)
5	+32	Y+ (Y+ slew)

6	+64	Z- (Z- slew)
7	+128	Z+ (Z+ slew)
8	+256	A- (A- slew)
9	+512	A+ (A+ slew)
10	+1024	B- (B- slew)
11	+2048	B+ (B+ slew)

A=REPORTTTLVALUES returns the current values for programmable TTL I/O ports

This is used to provide a snapshot IO port and slew input values. The report value is mapped identically to the TTL I/O commands (such as 'C' and 'D')

Bit	***Value***	***Signal***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	BO1
28	+268435456	BO2

REQUESTMOTORPAUSE pauses the vector system

RequestMotorPause is used to pause the vector system. It is completely identical to the PauseMotors command, and exists for compatibility of commands with the StepperBoard class library.

Motors may be restarted using the RequestMotorRestart function, or by issuing the STOPALLMOTORS or STOPVECTORMOTORS function (which flushes the queue and removes the pause flag).

```
RequestMotorPause
```

Starts the motors slowing down for the pause action.

REQUESTMOTORRESTART restarts a paused vector system

RequestMotorRestart is used to restart a paused system (which was paused by any of the motor pause events, such as a limit event, RequestMotorPause, or PauseMotorsWithTTLRestart).

It is ignored if the motors are not currently paused.

RESETBOARD(<ResetOption>) resets the board per the requested option

ResetBoard restarts the board (and aborts any running script!), using the <ResetOption> to control how the board does the restart.

ResetOption values are:

- -1: Strictly use factory default settings
- 0: Restart using EEProm saved settings
- >0:Restart using EEProm saved settings, but also force the microstep size (on the BC6D20 and BC6D25) or step pulse width (on the SD6DX) to the value of the ResetOption parameter

For example,

```
ResetBoard(2)
```

Would restart the board using a step pulse width of 2 microseconds on the SD6DX unit. It would also abort the script program…

A= SENSETRANSMITEMPTY

This function returns a non-0 value if the serial transmit buffer for the currently selected serial port is fully empty (i.e., all serial data has been sent). It returns a 0 if there is still data in the process of transmission.

SETADDRESSMODE(<AddressMode>

This sets the address mode being used for all vector goto actions as being absolute or current location relative. <AddressMode> defines the new mode: 0 (false) states motion is relative, while non-0 (true) states motion is absolute.

For example,

```
SetAddressMode(1)
```

Sets all motion to be absolute, which is the default.

SETCURRENTCOORDINATES(<W>,<X>,<Y>,<Z>,<A>,<B>) Defines the current location for all motors

SetCurrentCoordindates queues a reset of the defined motor locations for all motors. When this point in the motion queue is reached, then the current pending motor values become the new current locations, as opposed to being a new GoTo target.

For example, to reset the current locations on all motors to be defined as location 0, issue the command:

```
SetCurrentCoordinates(0,0,0,0,0,0)
```

SETCURRENTWHENMOTORIDLE(<idMotor>, <Current>) defines motor current to use when the motor is idle

SetCurrentWhenMotorIdle is used to define the motor current for the selected motors when they are idle. The <Current> is expressed in milliamps if the variable is integer, and in amps if the variable is floating point.

For example, to disable all motors when idle, issue the command

```
SetCurrentWhenMotorIdle(63, 0)
```

SETCURRENTWHENMOTORINMOTION(<idMotor>, <Current>) defines motor current to use when the motor is in motion

SetCurrentWhenMotorInMotion is used to define the motor current for the selected motors when they are in use. The <Current> is expressed in milliamps if the variable is integer, and in amps if the variable is floating point.

For example, to set all motors to ½ amp when in motion, issue the command

```
SetCurrentWhenMotorInMotion(63, 500)
```

SETDAC(<idDac>, <Value>) sets selected dac(s) to the requested value on the SD6DX

SetDac is used to set the requested dacs to the indicated value on the SD6DX board (it is an ignored command on the BC6Dxx products).

<idDac> is mapped as:

0	Dac 0 (V0)
1	Dac 1 (V1)

The dac value units are either millivolts or volts, depending on the parameter passed. If the parameter is integer, the dac value is expressed in millivolts. If it is floating point, then the value is expressed in volts.

For example,

```
SetDac(0, 400)
SetDac(1, 0.4)
```

Would actually set both dacs to 0.4 volts.

SETDACINMILLIVOLTS(<idDac>, <Value>) sets selected dac(s) to the requested value on the SD6DX

SetDacInMillivolts is used to set the requested dacs to the indicated value on the SD6DX board (it is an ignored command on the BC6Dxx products).

<idDac> is mapped as:

0	Dac 0 (V0)
1	Dac 1 (V1)

The dac value units are millivolts for this call.

For example,

```
SetDacInMillivolts(0, 400)
```

Would set dac 0 to 0.4 volts.

SETDACINVOLTS(<idDac>, <Value>) sets selected dac(s) to the requested value on the SD6DX

SetDacInVolts is used to set the requested dacs to the indicated value on the SD6DX board (it is an ignored command on the BC6Dxx products).

<idDac> is bit mapped as:

0	Dac 0
1	Dac 1

The dac value units are volts for this call.

For example,

```
SetDacInVolts(1, 1)
```

Would set dac 1 to 1 volt.

SETEXTDAC(<idDac>, <Value>) sets selected external ISODAC to the requested value

SetExtDac is used to set the requested external ISODAC to the indicated value on the any of our products.

<idDac> is mapped as:

0	ISODAC Board 0, set value, do not memorize
1	ISODAC Board 0, set value, memorize it for next power cycle
2	ISODAC Board 1, set value, do not memorize
3	ISODAC Board 1, set value, memorize it for next power cycle
4	ISODAC Board 2, set value, do not memorize
5	ISODAC Board 2, set value, memorize it for next power cycle
6	ISODAC Board 3, set value, do not memorize
7	ISODAC Board 3, set value, memorize it for next power cycle

The dac value units are either millivolts or volts, depending on the parameter passed. If the parameter is integer, the dac value is expressed in millivolts. If it is floating point, then the value is expressed in volts.

For example,

```
SetExtDac(0, 400)
SetExtDac(2, 0.4)
```

Would actually set both dacs to 0.4 volts.

SETEXTDACINMILLIVOLTS(<idDac>, <Value>) sets selected external ISODAC dac(s) to the requested value

SetExtDacInMillivolts is used to set the requested external ISODAC to the indicated value on the any of our products.

<idDac> is bit mapped as:

0	ISODAC Board 0, set value, do not memorize
1	ISODAC Board 0, set value, memorize it for next power cycle
2	ISODAC Board 1, set value, do not memorize
3	ISODAC Board 1, set value, memorize it for next power cycle
4	ISODAC Board 2, set value, do not memorize
5	ISODAC Board 2, set value, memorize it for next power cycle
6	ISODAC Board 3, set value, do not memorize
7	ISODAC Board 3, set value, memorize it for next power cycle

The dac value units are millivolts for this call.

For example,

```
SetExtDacInMillivolts(3, 400)
```

Would set ISODAC board 1 to 400 millivolts, and memorize the value.

SETEXTDACINVOLTS(<idDac>, <Value>) sets selected external ISODAC dac(s) to the requested value

SetExtDacInVolts is used to set the requested external ISODAC to the indicated value on the any of our products.

<idDac> is bit mapped as:

0	ISODAC Board 0, set value, do not memorize
1	ISODAC Board 0, set value, memorize it for next power cycle
2	ISODAC Board 1, set value, do not memorize
3	ISODAC Board 1, set value, memorize it for next power cycle
4	ISODAC Board 2, set value, do not memorize
5	ISODAC Board 2, set value, memorize it for next power cycle
6	ISODAC Board 3, set value, do not memorize
7	ISODAC Board 3, set value, memorize it for next power cycle

The dac value units are volts for this call.

For example,

```
SetExtDacInVolts(0, 2.5)
```

Would set ISODAC board 0 to 2.5 volts.

SETENCODERCOORDINATE(<Encoder0>, <Encoder1>) defines the current values for both encoders

SetEncoderCoordinate is used to define the current values for both encoders.

For example,

```
SetEncoderCoordinate(100,200)
```

Would set encoder 0 to a value of 100, and encoder 1 to a value of 200.

SETIOANDSLEWPORTDIRECTIONS(<IOPortDirections>, <SlewPortDirections>) sets the I/O direction for the IO and Slew ports

SetIOAndSlewPortDirections is used to set the I/O directions fot he 5 extended I/O signals and the SLEW ports. If a bit is set in the command, the direction is output. If it is clear, then the direction is input.

IOPortDirections is formatted as:

Bit	**Value**	**Use**
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2

SlewBits is formatted as:

Bit	**Value**	**Use**
0	+1	W-
1	+2	W+
2	+4	X-
3	+8	X+
4	+16	Y-
5	+32	Y+
6	+64	Z-
7	+128	Z+
8	+256	A-
9	+512	A+
10	+1024	B-
11	+2048	B+

To define W+ and NXT as outputs with all of the rest inputs, you would use the command

```
SetIOAndSlewPortDirections(2, 1)
```

SETIOANDSLEWPORTS(<IOBits>, <SlewBits>) Set the various I/O ports

SetIOAndSlewPorts sets the values for ALL of the programmable output bits. <IOBits> defines the extended IO lines, while <SlewBits> defines the SLEW lines.

IOBits is formatted as:

Bit	Value	Use
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2

SlewBits is formatted as:

Bit	Value	Use
0	+1	W-
1	+2	W+
2	+4	X-
3	+8	X+
4	+16	Y-
5	+32	Y+
6	+64	Z-
7	+128	Z+
8	+256	A-
9	+512	A+
10	+1024	B-
11	+2048	B+
12	+4096	WO1
13	+8192	WO2
14	+16384	XO1
15	+32768	XO2
16	+65536	YO1
17	+131072	YO2
18	+262144	ZO1
19	+524288	ZO2
20	+1048576	AO1
21	+2097152	AO2
22	+4194304	B01
23	+8388608	B02

For example,

```
SetIOAndSlewPorts(1, 4096)
```

Would set RDY and W01 high, and all other bits low.

SETIOANDSLEWPORTSHIGH(<IOBits>, <SlewBits>) sets the various I/O ports high

SetIOAndSlewPortsHigh sets matching output port bits high. The format for IOBits and SlewBits matches that of SetIOAndSlewPorts.

For example,

```
SetIOAndSlewPortsHigh(1, 4096)
```

Would set RDY and W01 high, leave all of the other bits unchanged.

SETIOANDSLEWPORTSLOW(<IOBits>, <SlewBits>) sets the various I/O ports low

SetIOAndSlewPortsLow sets matching output port bits low. The format for IOBits and SlewBits matches that of SetIOAndSlewPorts.

For example,

```
SetIOAndSlewPortsLow(1, 4096)
```

Would set RDY and W01 low, leave all of the other bits unchanged.

SETIOCONFIG(<Config>) configures special board features

The SetIOConfig function is used to configure the RDY and NXT lines for special operations, SI2/SO2 for TTL-serial, and encoder enables for the two on board encoder systems.

Please see the "t" command for details of the format of the <Config> parameter to this function.

For example, to set SI2 and SO2 up as TTL-Serial lines with all of the other features disabled, issue the command

```
SetIOConfig(32)
```

SETIOPORTDIRECTIONS(<PortDirections>) Sets all programmable TTL I/O line port directions.

SetIOPortDirections is used to define the I/O directions for all of the programmable I/O ports.

This command is bit-encoded identically to the SetIOPorts Command. Any bit with a value of '1' defines the associated port as being an output port. Any bit with a value of '0' defines that port as being an input port

The encoding of <PortDirections> is therefore:

Bit	***Value***	***Signal: 1 = output, 0 = input***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+

For example, to define NXT on the BC6D20 as an output port, issue the command

SetIOPortDirections(2)

To define W+ and NXT as outputs, you sum the 'values' for NXT and W+ (2 and 64 above, respectively), giving you the command:

SetIOPortDirections (66)

Please note: the low 5 bits of I/O (RDY, POT, SI2 and SO2) must have been defined as 'TTL I/O ports' through use of the 't' command before they can be programmed using the 'u' command.

SETIOPORTS(<BitValues>) Sets all I/O ports to the requested values

SetIOPorts is used to set all programmable output ports to the request values.

The <BitValues> parameter is bit encoded as:

Bit	***Value***	***Signal***
0	+1	RDY
1	+2	NXT
2	+4	SI2
3	+8	SO2
4	+16	IO2
5	+32	W-
6	+64	W+
7	+128	X-
8	+256	X+
9	+512	Y-
10	+1024	Y+
11	+2048	Z-
12	+4096	Z+
13	+8192	A-
14	+16384	A+
15	+32768	B-
16	+65536	B+
17	+131072	WO1
18	+262144	WO2
19	+524288	XO1
20	+1048576	XO2
21	+2097152	YO1
22	+4194304	YO2
23	+8388608	ZO1
24	+16777216	ZO2
25	+33554432	AO1
26	+67108864	AO2
27	+134217728	BO1
28	+268435456	BO2

Add the decimal value for each bit that you want to set to '1' (high), and issue the command on the sum. Those bits will get set: the others will be set to low.

For example, to set W01 high (on the SD6DX board) with all others low, issue the command

    SetIOPorts(131072)

SETIOPORTSHIGH(<BitValues>) Sets all selected I/O port bits high

SetIOPortsHigh is used to set all programmable output ports high that are set in the parameter passed.

The <BitValues> parameter is bit encoded as in the SetIOPorts function (above).

For example, to set RDY and IO2 high with all others untouched, issue the command

    SetIOPortsHigh(17)

SETIOPORTSLOW(<BitValues>) Sets all selected I/O port bits low

SetIOPortsLow is used to set all programmable output ports low that are set in the parameter passed.

The <BitValues> parameter is bit encoded as in the SetIOPorts function (above).

For example, to set RDY and IO2 low, with all others untouched, issue the command

    SetIOPortsHigh(17)

SETLIMITSWITCHENABLES(<idMotor>, <LimitOptions>)

SetLimitSwitchEnables is used to control interpretation of the board limit and slew switch inputs, as well as control of the motor output drivers on a per motor basis. The command includes control of whether or not the motor is treated as a "vector" motor or an "asynchronous" motor. **Changing this bit is a critical event** – it must not be changed while there is any motor motion occurring, since the code cannot reconcile vector and asynchronous mode changes while the motor or vector is active. On the SD6DX product, the firmware actively aborts motion if this function is called.

Bits	Action
0-2	Control LIMIT- input actions 2 1 0 0 0 0 (+0) Limit enabled, LOW stops (default) 0 0 1 (+1) Limit enabled, HIGH stops 0 1 0 (+2) Limit instant stop enabled, LOW stops 0 1 1 (+3) Limit instant stop enabled, HIGH stops 1 x x (+4) Limit disabled
3-5	Control LIMIT+ input actions 5 4 3 0 0 0 (+0) Limit enabled, LOW stops (default) 0 0 1 (+8) Limit enabled, HIGH stops 0 1 0 (+16) Limit instant stop enabled, LOW stops 0 1 1 (+24) Limit instant stop enabled, HIGH stops 1 x x (+32) Limit disabled
6	SLEW input disable 0 (+0) Enable slews (default) 1 (+64) Disable slews
7-9	Analog input channel for rate control 9 8 7 0 x x (+0) Analog rate control disabled 1 0 0 (+512) RDY is analog rate source 1 0 1 (+640) NXT is analog rate source 1 1 0 (+768) IO2 is analog rate source 1 1 1 (+896) <reserved>
10	Asynchronous or synchronous (vector) motor 0 (+0) Motor is operated via the VECTOR system 1 (+1024) Motor is operated asynchronously
11	SD6DX only O1 Output signal Manual Control bit 0 (+0) O1 is operated automatically 1 (+2048) O1 is operated via TTL output commands
12	SD6DX only O2 Output signal Manual Control bit 0 (+0) O2 is operated automatically 1 (+4096) O2 is operated via TTL output commands
13	SD6DX only Controls STEP signal polarity 0 (+0) Step is LOW-HIGH-LOW 1 (+8192) Step is HIGH-LOW-HIGH
14	SD6DX only Controls DIR signal polarity 0 (+0) Dir is LOW for minus 1 (+16384) Dir is HIGH for minus
15	BC6D25 only Controls whether a driver fault detected on this motor will abort all motion on the controller. 0 (+0) All motion is aborted on fault 1 (+32768) Do not abort motion on fault
16	Reassign '+' limit input to '-' limit input (allows single-line limit control. If set, the actual I/O port used to control the '+' limit input is reset to the same port as the '-' limit input. The interpretation of that input remains that as specified by bits 3-5 of this command. 2 (+0) +limit input is control of +limit actions 3 (+65536) –limit input is control of +limit actions

As is shown in the above table, the data value is encoded into multiple fields.

Two fields (LIMIT+ and LIMIT- settings) control the limit input behaviors associated with that motor (i.e., whether the limits are enabled at all, the sense level for stopping the motor, and whether the stop is a 'slow-down-and-stop' or an 'instant stop' action).

The next field controls whether the SLEW inputs are monitored for operating the motor when it is in asynchronous mode (slews are always ignored for vector motors).

The next field selects whether analog rate control is enabled for the motor (only when it is an asynchronous motor), and which input channel provides the rate voltage. See the 'v' command for details on configuring the mapping of the voltage input into a real rate.

The next field selects whether the motor is a vector motor or an asynchronous motor. Vector motors are controlled by the vector mode commands, while asynchronous motors are independently controlled by the asynchronous motor commands.

On the SD6DX boards, the following 2 fields control whether the O1 and O2 motor output signals are controlled automatically via the H and N commands, or manually via the TTL output commands.

On the SD6DX boards, the following 2 fields control the polarity of the STEP and DIR output signals.

On the SD6DX boards, the following 2 fields control whether the O1 and O2 motor output signals are controlled automatically via the H and N commands, or manually via the TTL output commands.

On the SD6DX boards, the following 2 fields control the polarity of the STEP and DIR output signals.

On the BC6D25 boards, the following field controls whether a “driver fault” will abort all motion (by default, all motion is aborted if a driver chip reports a problem).

By default, this parameter is set to 0. This enables the limit switches for low-true operation, and sets the motor as a 'vector' motor.

On all boards, the next bit controls reassignment of the ‘-’ limit input as control of the ‘+’ limit input actions (single-line limit control). If enabled, this allows the ‘-’ limit input to act as the limit source for both directions of motion, and is most useful for rotational ‘home’ limit actions. Additionally, when this feature is enabled, the associated ‘+’ limit input is effectively ignored for motion control; however, it is still reported in all reports that are appropriate (such as the ‘5?’, ‘6?’ and ‘9?’ reports).

By default, this bit is set to 0 and the feature is disabled.

SETMICROSTEPSIZE(<idMotor>, <MicroStepSize>) sets microstep size for the selected motors

On the BC6D20/BC6D25, the SetMicroStepSize command allows the microstep size to be separately specified for each pair of motors (as selected by the <idMotor> parameter). As with the reset command, the <MicroStepSize> must be one of the perfect powers of 2 from 1 to 16 or 32 (1, 2, 4, 8, 16, 32), and is expressed in units of 1/16th of a step on the BC6D20 and in units of 1/32nd of a step on the BC6D25.

Due to hardware limitations (i.e, we ran out of signals), the microstep size is assigned to each odd/even pair of motors. That is to say, motor pairs (W,X), (Y,Z) and (A,B) always have matching microstep sizes. If a differing microstep size is requested for two motors in a pair, the last one specified is used on both motors.

WARNING! Specification of the microstep size is a 'major event' for the board. You will get incorrect operation if you issue this command while any kind of motion is under way. Also, any non-0 locations will no longer have much meaning, since their units will have changed.

On the BC6D20, 1/8th step motion has a limit of 250,000 microsteps/second, while all other step sizes support 500,000 microsteps/second.

On the BC6D25, the maximum step rate limit is always 250,000 microsteps/second.

Note: The SD6DX firmware ignores this command.

SETMOTORBACKLASH(<idMotor>, <Backlash>) defines the motor backlash for the indicated motor(s)

SetMotorBacklash is used to specify (in microsteps) the motor backlash amount. The selected motor(s) (<idMotor>) all get their backlash set to the specified value. This value may be from 0 to 65535.

Whenever a motor's direction of spin changes, the motor's position gets preadjusted by the backlash amount so that the system will correctly wind-up (tighten) the backlash. This action is fully transparent to external code: it occurs at any time that a new spin direction is requested for each motor.

Note that the code does NOT insert a separate 'windup' vector to take up the backlash; rather, it increases the number of steps by the backlash amount whenever the motor's spin direction changes.

SETNEXTQUEUEELEMENTLONGID(<lNewValue>) sets the next queue element identifier to the requested value

SetNextQueueElementLongId is used to reset the queued element counter. This counter is a 32 bit incrementing value, that gets incremented after adding each new element to the queue. Note that it will 'wrap' when it increments from +2147483647: the next value will overflow to -2147483648.

The value that you specify as the parameter to SetNextQueueElementLongId is the one that will be assigned to the next queued element (for example, the next element generated by a 'G' command).

SETO1O2WHENMOTORIDLE(<idMotor>, <O1O2Setting>) defines the O1 and O2 outputs when the motor is idle

SetO1O2WhenMotorIdle defines the state of the motor's xO1 and xO2 lines when the motor is idle, assuming that they have been configured for automatic control.

O1O2Setting bit 0 is for xO1, while bit 1 is for xO2.

For example,

```
SetO1O2WhenMotorIdle(2, 1)
```

Would set O1 high and O2 low when the X motor is idle.

SETO1O2WHENMOTORINMOTION(<idMotor>, <O1O2Setting>) defines the O1 and O2 outputs when the motor is in motion

SetO1O2WhenMotorInMotion defines the state of the motor's xO1 and xO2 lines when the motor is in motion, assuming that they have been configured for automatic control.

O1O2Setting bit 0 is for xO1, while bit 1 is for xO2.

For example,

```
SetO1O2WhenMotorIdle(2, 2)
```

Would set O1 low and O2 high when the X motor is idle.

SETPOTRATESCALES(<idWhichMotor>, <MinRate>, <MaxRate>, <AToDAtMinRate>, <AToDAtMaxRate>) defines the rate mappings for potentiometer control for the indicated motor

SetPotRateScales is used to define the A/D (potentiometer) input scaling and limits for use when a potentiometer is used to control the rate for the selected motor.

<idWhichMotor> selects which set of motors are to receive the effects of the commands.

<MinRate> defines the minimum rate that is attainable using the A/D input system.

<MaxRate> defines the maximum rate that is attainable using the A/D input system.

<AToDAtMinRate> defines the raw A/D reading that is associated with the MinRate parameter, expressed such that 4095 maps into 3.3 volts.

<AToDAtMaxRate> defines the raw A/D reading that is associated with the MaxRate parameter, expressed such that 4095 maps into 3.3 volts.

Please note: you cannot attain a rate of 0 using this system. Such rate value will not be accepted.

For example,

```
SetPotRateScales(1, 40, 4000, 0, 4095)
```

Would define access for motor W as a mapping of the A/D reading of 0 volts into a rate of 40, and a reading of 3.3 volts (A/D raw value 4095) into a rate of 4000.

SETPOTRATESCALESTODEFAULTS(<idMotor>)

SetPotRateScalesToDefaults sets the default mappings for the rate scale system on the selected motors.

The defaults are that a 0 A/D reading maps into a rate of 1, while a full scale (4095) reading maps into a rate of 500,000 on the BC6D20 and Sd6DX units, and into 250,000 on the BC6D25 unit.

SETRAMPRATE(<RampRate>) defines the vector ramp rate

SetRampRate defines the ramp rate for vector motion, in microsteps/second/second.

```
SetRampRate(25000)
```

Would set the vector ramp rate to 25000 microsteps/second/second.

SETRATESCALEFACTOR(<Multiplier>)

SetRateScaleFactor is used to dynamically specify a fixed point multiplier to apply to the current target rate.

This factor may be applied at any time, including when motion is under way. The code will automatically ramp the current rate up and down accordingly in order to attain the requested new rate, through use of the 'ramp rate' parameter.

For example, if your current rate ("R") is set to 2000, issuing the command

```
SetRateScaleFactor(0.5)
```

will tell the system that the real rate is 0.5 * R, or 1000. If the motors are already moving faster than 1000, they will slow down to the 1000 value.

A special value of 0 turns off rate scaling -- this is identical in results to a value of 1.

SETREPORTQUEUEELEMENTSNAPSHOT(<idWhichElement>)

SetReportQueueElementSnapshot generates a queue snapshot report into the Snapshot array.

Values that are >= 0 (currently, values of 0 to 99) directly connect to the fixed array of queue values; this can be used to manually read any element of the queue (usually for diagnostic purposes).

There are four reserved values which report special 'queue' elements:

-1: Most recent element that has been (or is being) processed

-2: Prior GOTO element that was last queued

-3: Current data on GOTO element currently being built for queue

-4: Current location data for all motors, as an instant snapshot

The data is captured for the ReportQueueSnapshot function to use, as elements:

0: Sequence number for element in queue
  Note: on the -4 query, this will be the NEXT sequential
  number that will be used on the next queue element.
1: Internal command associated with element
  For real queue elements (>= 0),
  if the cmd value is positive, it has not yet been processed.
  If it is negative, then it has been pulled from the queue.
2: Floating point parameter for element
3 to 9: 32 integer parameters lp0 through lp6 associated with this element

The single element type that is of most interest is the one with the 'cmd' value being 1. This is a vector command, and assigns the remaining parameters as:

fparam: target rate ('R' command)
lp0: vector length (parameter to 'G')
lp1: W motor target
lp2: X motor target
lp3: Y motor target
lp4: Z motor target
lp5: A motor target
lp6: B motor target

SETREQUESTEDALOCATION(<TargetLocation>) Set the next vector target location for motor A

SetRequestedALocation sets the next vector target location for motor A.

    SetRequestedALocation(100)

Sets the next vector target location for motor A to 100.

SETREQUESTEDBLOCATION(<TargetLocation>) Set the next vector target location for motor B

SetRequestedBLocation sets the next vector target location for motor B

    SetRequestedBLocation(100)

Sets the next vector target location for motor B to 100.

SETREQUESTEDLOCATION(<W>,<X>,<Y>,<Z>,<A>,<B>) Sets the next vector target location for all motors

SetRequestLocation sets the next vector target location for all motors (W, X, Y, Z, A, and B).

    SetRequestedLocation(100,200,300,400,500,600)

Sets the next target vector location to be W=100, X=200, Y=300, Z=400, A=500 and B=600.

SETREQUESTEDWLOCATION(<TargetLocation>) Set the next vector target location for motor W

SetRequestedWLocation sets the next vector target location for motor W.

    SetRequestedWLocation(100)

Sets the next vector target location for motor W to 100.

SETREQUESTEDXLOCATION(<TargetLocation>) Set the next vector target location for motor X

SetRequestedXLocation sets the next vector target location for motor X.

    SetRequestedXLocation(100)

Sets the next vector target location for motor X to 100.

SETREQUESTEDYLOCATION(<TargetLocation>) Set the next vector target location for motor Y

SetRequestedYLocation sets the next vector target location for motor Y.

    SetRequestedYLocation(100)

Sets the next vector target location for motor Y to 100.

SETREQUESTEDZLOCATION(<TargetLocation>) Set the next vector target location for motor Z

SetRequestedZLocation sets the next vector target location for motor Z.

    SetRequestedZLocation(100)

Sets the next vector target location for motor Z to 100.

SETRUNRATE(<RunRate>) defines the target run rate to use on the next vector

SetRunRate defines the target run rate to apply to the next vector to be generated.

    SetRunRate(2000)

Would set the target vector run rate to 2000 microsteps/second.

SETSTOPOKRATE(<StartStopRate>) defines the start/stop rate for the vector system

SetStopOKRate defines the start/stop rate used by the vector system

```
SetStopOKRate(80)
```

Would set the vector start/stop rate to 80 microsteps/second.

SETVERBOSEMODE(<Modes>) defines the serial communication rules

The SetVerboseMode command is used to control whether the board transmits a "<CR><LF>" sequence before it processes a command (in normal serial command mode as opposed to the scripting system), and whether a spacing delay is needed before any command response. **By default (after power on and after any reset action), the board is normally configured to echo a carriage-return, line-feed sequence to the host as soon as it recognizes that an incoming character is not part of a numeric value.** This allows host code to fully recognize that a command is being processed; receipt of the <LF> tells it that the command has started, while receipt of the final "*" states that the command has completed processing. 'V' is ignored if you fail to provide a parameter (i.e., 'V' alone results in the board ignoring the command, while '0V' causes the normal 'V' operation with a parameter of '0').

The firmware actually recognizes and responds each new command within a few microseconds of receipt of the stop bit of the received character. In most designs, this will not be a problem; however, since all commands issue an '*' upon completion, and they can also (by default) issue a <CR><LF> pair before starting, it is quite possible to start receiving data pertaining to the command before slower non-interrupt based devices can change their state! In microprocessor, non-buffering designs (such as with the Parallax, Inc.^{tm} Basic Stamp ^{tm} series of boards), this can be a significant issue. The firmware handles this via a configurable option in the 'V' command. If enabled, the code will delay 1 millisecond upon receipt of a new command character. This really means that the first data bit of a response to a command will not occur until at least 9 bit intervals after completion of transmission of the stop bit of that command (about 900 uSeconds at 9600 baud); for the Basic Stamp^{tm} this is quite sufficient for it to switch from send mode to receive mode.

The verbose command is bit-encoded as follows:

Bit	SumValue	Use When Set
0	+1	Send <CR><LF> at start of processing a new command
1	+2	Delay about 1 millisecond before transmission of first character of any command response.
2	+4	Send numeric responses in hexadecimal instead of decimal.
3-4		<reserved: ignored>
5	+32	If set, code always sends responses: incoming characters do NOT stop pending data. WARNING -- This can cause loss of incoming commands, if 4 or more characters sent while the 64 character transmit buffer is full!
6	+64	If set, sending a 'CTRL-C' character (hex 0x03) will flush any pending input data that has not yet been consumed by the firmware, leaving just the CTRL-C character as the next character to be read. This mode is useful to allow immediate aborts of running scripts.

If you set verbose mode to 0, then the <CR><LF> sequence is not sent. Reports still will have their embedded <CR><LF> between lines of responses; however, the initial <CR><LF> which states that the command has started processing will not occur.

For example,

```
SetVerboseMode(0)
```

would block transmission of the <CR><LF> command synch, and could respond before completion of the last bit of the command, while

```
SetVerboseMode(3)
```

would enable transmission of the <CR><LF>sequence, preceded by a 1-character delay.

STARTMULTIVECTORSEQUENCE(<lValue>) starts (or ends) a multi-vector sequence

StartMultiVectorSequence is used to control how a series of vectors get joined, in terms of rate ramping.

The parameter passed normally defines the total length for the vector (the parameters to the following 'GotoLocation…' commands), so that the code can correctly treat the ensuing motion requests as one trapezoidal motion event. The firmware automatically adjusts this length for any motor backlash encountered as the motion progresses. The motion is considered to be complete when this total distance has been consumed by the pending vectors (note: the last vector may extend the total distance if the 'M' value is too small).

There are 3 possible sets of values for the StartMultiVectorSequence lValue parameter:

> 0: This is the total vector length for the entire set of motion events
= 0: Complete any merged motion here – next 'G' starts a new event
= -1: Start automatic chained vector mode.

The final mode of '-1' allows for mostly automatic merging of vectors by the code. In this mode, the firmware will keep 'chaining' successive vectors together until a new StartMultiVectorSequence command is issued (or a quit is done to abort all motion). If you are not able to 'feed' vectors to the queue rapidly enough to stay ahead of the motion, the code will automatically slow down and speed up the motion as needed to prevent any sudden stops from happening if the queue gets emptied. The '-1' mode should always be completed with either a '0' or a '-1' trailing StartMultiVectorSequence command (thus starting a new vector), to fully optimize the generated motion.

The >0 mode is far less forgiving. It requires that you keep the queue from emptying by sending data rapidly enough to stay ahead of the motion. If the firmware runs out of motion events while processing a set of vectors when in this mode, the motors will experience an instant stop, followed by an attempt to perform an instant start (at whatever rate had been attained by the time of the instant stop) as soon as another vector makes it into the queue. The effect is likely to be very choppy motion and missed steps! This mode is present for "compatibility" with the NC firmwares on earlier products, and is not recommended for current use.

For example,

```
GoToLocation(0,0,0,0,0,0)
StartMultiVectorSequence(-1)   ` Start a multi-vector sequence
GoToLocation(100,100,0,0,0,0)
GoToLocation(150,200,0,0,0,0)
StartMultiVectorSequence(0)
```

Would draw the second 2 vectors with just one trapezoidal motion sequence, for smoother operation.

STOPALLMOTORS stops motion on all motors now

StopAllMotors aborts all motion on all motors immediately with no ramping actions. **Please note that this action can damage gear systems, since the stop is immediate, with no slow-down behaviors!**

The queue is cleared, and the motors get set to their 'idle' state.

```
StopAllMotors
```

Issues the emergency stop event.

STOPASYNCHMOTORS stops motion on asynch motors

StopASynchMotors is identical to StopAllMotors, except that it only affects motors that are operating in the asynchronous mode. **Please note that this action can damage gear systems, since the stop is immediate, with no slow-down behaviors!**

```
StopASynchMotors
```

Will stop any motor that is operating asynchronously. Vector motion would continue.

STOPVECTORMOTORS stops all vector motion now

StopVectorMotors is identical to StopAllMotors, except that only the vector motion is affected. **Please note that this action can damage gear systems, since the stop is immediate, with no slow-down behaviors!**

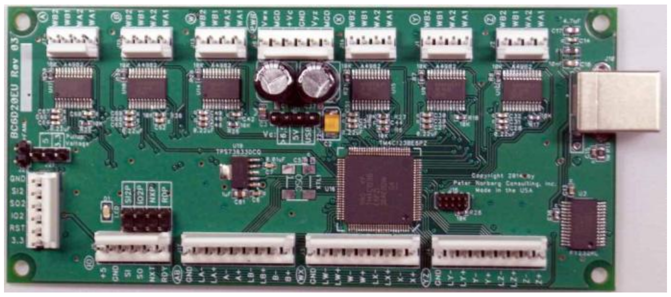
```
StopVectorMotors
```

Would therefore just stop vector motion. Any asynchronous motion would continue.

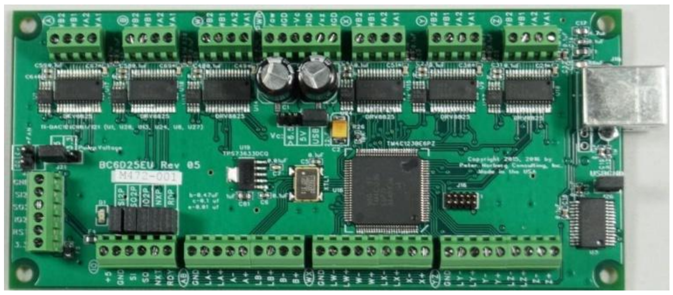
Board Hardware

Board Connections

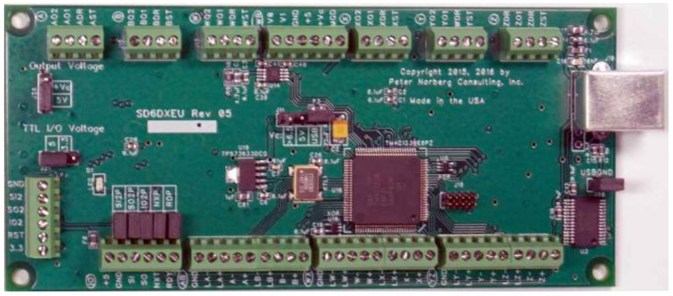
An example of the BC6D20 revision 3 board is shown below. The BC6D25 has identical connections, but it has a few extra jumpers. The SD6DX has identical appearing connectors, but the signal definitions on the top side (motor and power) are quite different.



BC6D20 revision 3 (with Amp MTA-100 Connectors)



BC6D25 revision 5 (with screw terminal Connectors)



SD6DX, Revision 5 (with screw terminal connectors)

Board Size

The BC6D20, BC6D25 and SD6DX boards, oriented as shown on this page, are 4.9 inches wide by 2.25 inches high.

Mounting Requirements

The mounting holes are 0.125 inches in diameter, and are positioned exactly 0.125 inches in from each corner. They allow up to a number 5 screw, which thus allows use of the standard #4 mounting spacers. Use washers that are 0.25" or smaller in output diameter, to avoid damage to board components that are close to the holes. Horizontally, their centers are 4.65 inches apart on the BC6D20, and vertically they are 2.00 inches apart. Thus, when the board is positioned as shown above, their positions are:

(0.125, 0.125), (4.775, 0.125),
(0.125, 2.125), (4.775, 2.125)

Connector Signal Pinouts

On the left side, we have:

- BC6D20/BC6D25: Fan power connector
- SD6DX: Motor connector output voltage selector jumper (5 volts or Vc)
- 5 volt/3.3 volt I/O logic power selector jumper
- Extra TTL IO logic connector (GND, SI2, SO2, IO2, RST, 3.3V) (on leftmost side)

Going from bottom-left to the right, we have:

- IO logic connector (+5, GND, SI, SO, NXT, RDY) (on leftmost side)
- I/O Pullup option jumpers SI2P, IO2P, NXP and RDP (directly above IO logic connector). On the SD6DX, an additional SOP jumper position may be present. Some boards may also have the SO2P pullup as well.
- Motor A, B TTL control: Limits and slews for motors A and B
- Motor W, X TTL control: Limits and slews for motors W and X
- Motor Y, Z TTL control: Limits and slews for motors Y and Z

On the right side we have:

- USB connector. On the SD6DX and BC6D25, we also may have the USBGND isolation jumper.

Going from top-left to the right, we have:

- A Motor connector (upper left)
- B Motor connector
- W Motor connector
- Power connector (center: provides separate motor and logic power)
- X Motor connector
- Y Motor connector
- Z Motor connector (upper right)

Extra TTL Logic Connector

The extra logic connector for the all boards, with a pinout of:

Pin	**Name**	**Description**
1	GND	Ground
2	SI2	SI2 extended TTL signal
3	SO2	SO2 extended TTL signal
4	IO2	IO2 generic TTL signal
5	RST	Board reset
6	3.3	3.3 volt power reference

It provides access to the SO2, SI2, IO2 and RESET signal. See the ‘C’, ‘D’, ‘T’, and the ‘6?’ and ‘7?’ commands for access to the SO2 and SI2 lines.

Board status and TTL Serial

Name	**Description**
+5	Access to +5 from the board
GND	Ground reference for all signals
SI	INPUT: Raw Serial Input (TTL level)
SO	OUTPUT: Raw Serial Output (TTL Level)
NXT	NXT generic I/O
RDY	Ready/busy output

This connector gives access to the serial control signals for the microprocessor, as well as board status and slew rates.

NXT is often used to act as a 'global instant stop' for all motors. It can also be used as a generic TTL I/O signal, as well as an analog (0 to 3.3 volt) input.

RDY is normally an informational output that describes the state of "one or more motors are still stepping". High means READY/IDLE, low means STEPPING.

SI and SO are the serial input and serial output (respectively), as seen by the microprocessor chip.

The default communication rate is normally set to 9600 baud, no parity, 8 data bits, 1 stop bit.

Firmware Factory Reset – Short SI and SO together

The SI and SO pins have a special extra function which may be used if you have set options in the board which make it unresponsive (such as setting the baud rate to a value that you have forgotten). If you short those two pins together (using a standard shunt), the firmware will reset itself to its default values (including the standard 9600 baud communications setting). Motor control is disabled as long as those pins are shorted; so your process for performing a firmware reset is:

1. Turn off power to the board
2. Disconnect all wires from the board except for power
3. Short SI and SO together
4. Apply power to the board for at least 5 seconds (wait until the LED illuminates, if you are powering the board off of the USB connector)
5. Turn off power (or unplug from the USB system)
6. Remove the SI-SO short
7. Reconnect your board as usual: it should now be at its factory default settings.

Pullup Option Jumpers

The option jumpers positioned just above the IO connector are used to control whether their associated signals are connected through a resistor to the currently selected pullup voltage (3.3 or 5 volts).

On the SD6DX unit, there may be an SOP (for SO Pullup) which must always be installed for normal operation. It is present for factory testing operations.

For IO2P (IO2 pullup), NXP (NXT pullup) and RDP (RDY pullup), you will normally have the jumpers installed at all times UNLESS the selected signal is to be used as an analog input. For those three signals, the jumper must be removed if the signal is to be used as analog.

For SI2P (SI2 pullup), you will have the jumper removed if you are using TTL serial, and installed if you are using it as either normal TTL input or output.

If it is available, SO2P (SO2 pullup), you will have the jumper installed if you are using TTL serial, and it will be installed or removed as needed if you are using it as either normal TTL input or output.

Motor A and B TTL I/O

Name	**Description**
GND	Ground reference for inputs – short input to GND to denote limit
LA-	A Minimum limit reached
LA+	A Maximum limit reached
A-	Slew A motor in minus direction
A+	Slew A motor in plus direction
LB-	B Minimum limit reached
LB+	B Maximum limit reached
B-	Slew B motor in minus direction
B+	Slew B motor in plus direction

The Motor AB TTL I/O connector provides for the TTL control of the A and B motors. All of the signals are pulled up (to the pullup voltage of 3.3. or 5 volts) through 10K resistors.

Motor W and X TTL I/O

Name	**Description**
GND	Ground reference for inputs – short input to GND to denote limit
LW-	W Minimum limit reached
LW+	W Maximum limit reached
W-	Slew W motor in minus direction
W+	Slew W motor in plus direction
LX-	X Minimum limit reached
LX+	X Maximum limit reached
X-	Slew X motor in minus direction
X+	Slew X motor in plus direction

The Motor WX TTL I/O connector provides for the TTL control of the W and X motors. All of the signals are pulled up (to the pullup voltage of 3.3. or 5 volts) through 10K resistors.

Motor Y and Z TTL I/O

Name	**Description**
GND	Ground reference for inputs – short input to GND to denote limit
LY-	Y Minimum limit reached
LY+	Y Maximum limit reached
Y-	Slew Y motor in minus direction
Y+	Slew Y motor in plus direction
LZ-	Z Minimum limit reached
LZ+	Z Maximum limit reached
Z-	Slew Z motor in minus direction
Z+	Slew Z motor in plus direction

The Motor YZ TTL I/O connector provides for the TTL control of the Y and Z motors. All of the signals are pulled up (to the pullup voltage of 3.3. or 5 volts) through 10K resistors.

Motor connectors A, B, W, X, Y Z on the BC6D20 and BC6D25

On the BC6D20/BC6D25, the connectors are for direct connection to the motor wires.

Name	**Description**
WA1	Winding A side 1
WA2	Winding A side 2
WB1	Winding B side 1
WB2	Winding B side 2

All six motor connectors are identical, and are used to connect to the bipolar motors as needed. Please see the section "Wiring Your Motor" on how to do this wiring.

Motor connectors A, B, W, X, Y Z on the SD6DX boards, wire to external step-and-direction drivers

On the SD6DX, the connectors are for signals that you connect to your step-and-direction driver board.

Name	**Description**
xST	Step pulses to your Step And Direction driver board
xDR	Direction signal to your Step And Direction driver board
xO1	Generic output 1: Often used to control the current of your Step And Direction board
xO2	Generic output 2: Often used to control the current of your Step And Direction board

All six motor connectors are identical, and are used to connect to the step and direction driver boards as needed.

Please note: *On releases of the SD6DX up through revision 6, a silkscreen error incorrectly labeled the 'Z' connector with two "ZDR" pins. The ZDR that is by the 'W' label should be labeled as "ZO2". The labeling should be as follows:*

Incorrect	Correct
ZST	ZST
ZDR	ZDR
ZO1	ZO1
ZDR	**ZO2**

On later shipments, a paper label has been applied to the board to correct this error.

These connectors give access to the actual step-and-direction signals as generated by the board. These pins are outputs of an on-board ULN2803 buffer driver with 1K pullups installed, for compatibility with high-current devices (such as the Gecko series of motor drivers).

By default, these outputs are configured to be compatible with most step-and-direction motor drivers. The 'direction' signals define the direction of motor motion each time that an associated 'step' pulse event occurs.

The outputs are configured in terms of polarity (high or low true) through use of the 'T' command.

The direction signals all default to 'high' means 'negative', 'low' means 'positive' (in terms of the motion as modeled by the firmware); the 'T' command may be used to invert these definitions.

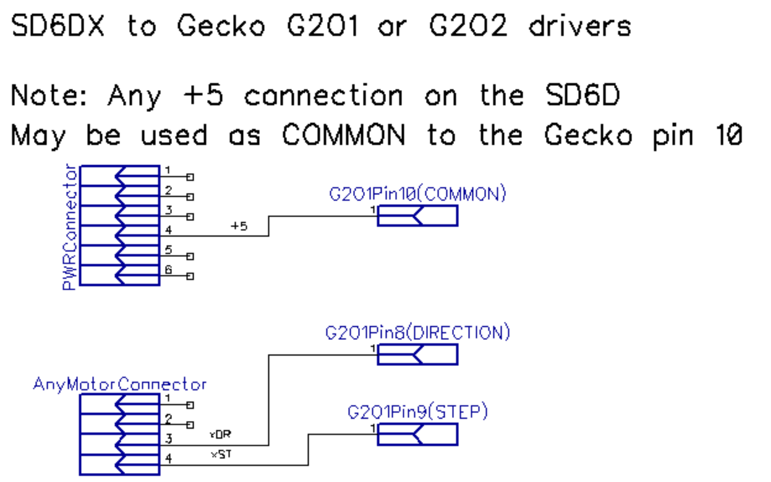
Similarly, the step signals all default to operate as 'low-high-low'; that is to say, a 'pulse' is defined as a signal which starts as a TTL low, goes high for an indicated number of microseconds (as defined by the '!' command), and then returns high. The 'T' command may be used to reverse this definition.

Many step-and-direction boards use optically isolated inputs to buffer the incoming signals. You will have to refer to the manufacturer's instructions to determine how to wire our 0-5 volt TTL signals to those boards. In most cases, our GND signal will be connected to their 'common', while our signal lines are connected to the appropriate step or direction inputs. In some cases (such as with the Gecko drivers version G2O1 and G2O2), our +5V will need to be connected to their 'common' (due to the way that they have wired their optical isolators); our step and direction signals will still be connected to their appropriate signal inputs as needed.

You will then have to determine whether the polarity of our step pulses is correct: an incorrect choice can cause a missed step when a direction change is done, since the direction signal will be changed during the wrong state of the 'pulse'. Again, you will have to study the motor driver's manual to determine the correct settings; feel free to call us with questions, but we will probably need access to a copy of your documentation in order to be able to give you definitive answers.

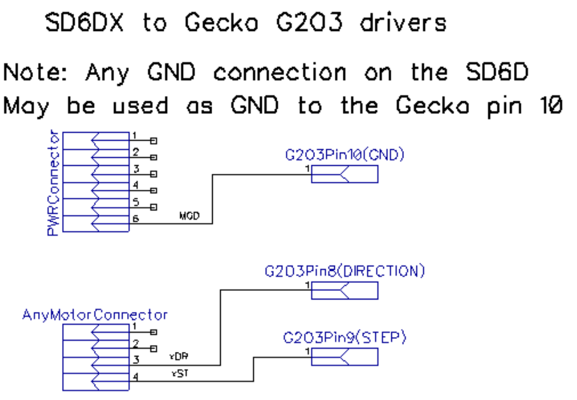
SD6DX Connection Example to the Geckodrive G2O1 and G2O2 drivers

The following schematic shows the connection from any one of the 6 driver outputs on the SD6DX board to a G2O1 or G2O2 driver. Please note that the G2O1 and G2O2 will probably require you to use the 'T' command to invert the polarities of the output signals, since they are HIGH-LOW-HIGH logic while the board defaults to LOW_HIGH_LOW logic.


SD6DX to Gecko G2O1 or G2O2 drivers
Note: Any +5 connection on the SD6D
May be used as COMMON to the Gecko pin 10
PWRConnector
1
2
3
4
5
6
+5
G2O1Pin10(COMMON)
1
G2O1Pin8(DIRECTION)
1
AnyMotorConnector
1
2
3
4
xDR
xST
G2O1Pin9(STEP)
1

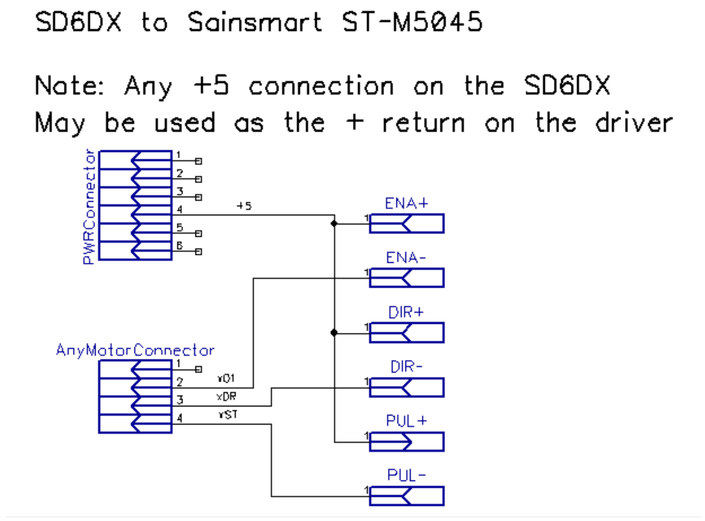
SD6DX Connection Example to the Geckodrive G2O3 driver

The following schematic shows the connection from any one of the 6 driver outputs on the SD6DX board to a G2O3 driver.


SD6DX to Gecko G2O3 drivers
Note: Any GND connection on the SD6D
May be used as GND to the Gecko pin 10
PWRConnector
1
2
3
4
5
6
MGD
G2O3Pin10(GND)
1
G2O3Pin8(DIRECTION)
1
AnyMotorConnector
1
2
3
4
xDR
xST
G2O3Pin9(STEP)
1

SD6DX Connection Example to the Sainsmart ST-M5045 driver

The following schematic shows the connection from any one of the 6 driver outputs on the SD6DX board to a Sainsmart ST-M5045 driver. Please note that the ST-M5045 will probably require you to use the 'T' command to invert the polarities of the output signals, since they are HIGH-LOW-HIGH logic while the board defaults to LOW_HIGH_LOW logic.


SD6DX to Sainsmart ST-M5045
Note: Any +5 connection on the SD6DX
May be used as the + return on the driver
PWRConnector
+5
AnyMotorConnector
xO1
xDR
xST
ENA+
ENA-
DIR+
DIR-
PUL+
PUL-

Power Connector And Motor Voltages, (DAC output on SD6DX)

On the BC6D20/BC6D25, the power connector appears as follows:

Name	***Description***
Vaw (or Vwx)	BC6D20: 7.5-34 volts, for the A, B and W motors (label error in the original BC6D20 silkscreen; should be Vaw) BC6D25: 10-40 volts.
MGD	Ground for Vaw
+Vc	Supply volts for the logic circuits. If the jumper near the power connector is in the '5V' position, then you must provide 5 volts here. If that jumper is in the ">6.5" position, then you must provide 6.5 to 15 volts here. If it is in the 'USB' position, then this signal is not connected.
GND	Ground for +Vc
Vxz (or Vyz)	BC6D20: 7.5-34 volts, for the X, Y and Z motors (label error in the original BC6D20 silkscreen; should be Vxz) BC6D25: 10-40 volts.
MGD	Ground for Vxz

There are two ways of powering the logic circuits for system, which can simplify selection of a power supply to use in driving the system.

If you are going to operate your motors with a supply from 7.5 through 15 volts on the BC6D20 or from 10 to 15 volts on the BC6D25, then one supply may be used to drive all of the motors and the logic supply. To do this, you would set the jumper near the power connector to its ">6.5" position, and then separately run #22 wires from each of the above pins to the power supply (i.e., Vaw, +Vc and Vxz each get separately run to the + output of power supply, while the two MGD and the GND signals each get separately run to the GND output of the power supply).

If you need to "split" the supplies (due to current limitations, or because you need to operate the motors at a value above 15 volts), then you would create what is called a "star ground". In this arrangement, all three of the ground wires from the power connector (the two MGD signals and the GND signal) get separately run with #22 wire to a common heavy-duty tie point, and all of your power supplies get run to that same tie point. Then you run the Vaw to its + supply, the Vxz to its + supply, and the Vc to its + supply. For the +Vc, you may elect to either use a supply from 6.5 to 15 volts (in which case you set the jumper near the power connector to the ">6.5" position), or you may use a 5 volt supply, and set that jumper to its "5V" position.

Be forewarned: if you have the +Vc jumper set to the "5V" position, and connect the "+Vc" to a supply which is not a 5 volt supply, you will destroy the board! This abuse is not covered by our warranty!

On the BC6D25, you may elect to "digitally isolate" the board if it is not having the logic portion powered off of the USB system. If the Power Options jumper is set to either the +5 or the >6.5 position, you may remove the "USBGND" jumper that is near the USB connector. Once this is done, the board and its power is fully isolated from the USB system (and its computer) using a digital isolator. This can avoid grounding issues on some installations.

Similarly, on the BC6D25, if you elect to power the board logic off of the USB system through having the Power Options jumper set to its USB position, then you MUST have the USBGND jumper installed. If you do not, then there will not be valid power available to the board from the USB system, and you are very likely to have ground and power problems.

On the SD6DX, the power connector appears as follows depending on artwork:

Original beta release (rev 01):

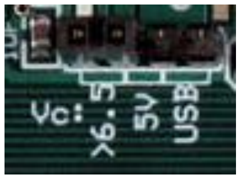
Name	***Description***
V0	0 to 5 volt DAC output V0 (see the 'J' command)
GND	Ground for V0
V1	0 to 5 volt DAC output V1 (see the 'J' command)
GND	Ground for V1
+Vc	Supply voltage for the logic circuits. If the jumper near the power connector is in the '5V' position, then you must provide 5 volts here. If that jumper is in the ">6.5" position, then you must provide 6.5 to 15 volts here. If it is in the 'USB' position, then this signal is not connected.
MGD	Ground for +Vc

First production-ready release (rev 04):

Name	***Description***
V0	0 to 5 volt DAC output V0 (see the 'J' command)
V1	0 to 5 volt DAC output V1 (see the 'J' command)
GND	Ground reference for the DAC outputs
+5	+5 reference for optical isolators (such as with many of the Gecko driver boards)
+Vc	Supply voltage for the logic circuits. If the jumper near the power connector is in the '5V' position, then you must provide 5 volts here. If that jumper is in the ">6.5" position, then you must provide 6.5 to 15 volts here. If it is in the 'USB' position, then this signal is not connected.
MGD	Ground for +Vc

Be forewarned: if you have the jumper set to the "5V" position, and connect the "+Vc" to a supply which is not a 5 volt supply, you will destroy the board! This abuse is not covered by our warranty!

The BC6D20, BC6D25 and SD6DX boards also have the option of powering the logic off of the USB system.



The 'USB' position enables powering the logic portion of the board from the USB 5 volt system. When the jumper is in the USB position, power is applied to most of the board logic only while the computer is 'awake' – that is to say, if the computer is turned off, asleep, or in some hibernation mode, then the board will act as if no power is present.

Note that the board draws about 80-150 mA (depending on use of the TTL signals) without the fan assembly, and as much as 350 mA with the fan assembly. This means that, for the USB-powered mode to work, the connection must be a high-power connection (either directly to the computer, or via a good powered hub).

On the SD6DX revision 5 and later boards, you may elect to "digitally isolate" the board if it is not having the logic portion powered off of the USB system. If the Power Options jumper is set to either the +5 or the >6.5 position, you may remove the "USBGND" jumper that is near the USB connector. Once this is done, the board and its power is fully isolated from the USB system (and its computer) using a digital isolator. This can avoid grounding issues on some installations.

Similarly, on the SD6DX, if you elect to power the board logic off of the USB system through having the Power Options jumper set to its USB position, then you MUST have the USBGND jumper installed. If you do not, then there will not be valid power available to the board from the USB system, and you are very likely to have ground and power problems.

The power options for all three products can be summarized as:

Motor Voltage	***PowerJumper***	***Number of power supplies***	***Comments***
6.5-15V	>6.5	1	Single power supply, connected with separate wires to each of the power connector pins (as described above)
5-34	>6.5	2-3	Use two or three power supplies, one for the logic (6.5 to 15 volts), one or two for the motors. Use a "star ground" power arrangement.
5-34	5V	2-3	Same as above, except that the logic supply is 5 volts.
5-34	USB	1-2	Vc power is via the USB system. Use one or two power supplies (with a "Star ground") to power the motors.

Board mounting and cooling considerations for the BC6D20 and BC6D25

Note that if the current requirements are over about 0.9 Amps/winding, or if you intend to leave the windings on when the motor is idle (see the 'N' command), or if the motor voltage supply exceeds 15 volts, then fan-based cooling of the board is normally required. The driver components can get quite hot, and external cooling will increase their lifetime considerably! You should also use fan-based cooling if you are going to be operating the board in a warm environment (>100 degrees F), or if the board is running "hotter" than you like.

The board itself acts as the heat sink for the motor driver chips: much of the heat dissipation is from the **bottom** of the board, so having adequate air flow and clearance is a requirement. For low current operation (less than 150 mA and 12 volts), there are usually few issues; otherwise, the minimum mounting clearance needed is 1/4 inch for medium currents (150 mA to 500 mA or cases where the windings are left on at reduced levels continuously), and ½ inch for larger draws or for any other continuous duty operations.

We strongly suggest that you use fan-based cooling if the current requirements exceed 0.9 amps per winding or if there is any evidence that the board is getting too hot.

Fan based cooling should be done such that the airflow is directed toward the top or bottom surface of the board, centered over the six motor driver chips (which is why a dual fan is needed). **Do not use a fan which blows air away from the board**; this is completely ineffective in terms of cooling the system. The fan assembly should provide about 8-12 CFM of air flow. Note that the board includes mounting holes positioned such that two 60 mm fans may be directly mounted, through use of four #4 standoffs (and assuming that you are willing to mount a fan with just supports along one edge). If the fans are mounted facing down at the top of the board (which cools the driver chips better), use 1 inch standoffs. If mounted facing up at the bottom of the board, use ½ inch standoffs. We offer a fan assembly that includes a bracket to tie the two fans together (for better support and stability). You may wish to review our section "Install the Fan Assembly", located at the end of this manual.

Calculating Current Requirements (BC6D20/BC6D25)

This section note describes how to calculate the power requirements for your motors, and for the system as a whole.

1. Determine the individual motor winding current requirements.

This can be determined by reading the specifications for your motor.

2. Determine current requirement for actually operating the motor(s)

Since our board microsteps, you may have points of operation wherein both motor coils are on at the same time. On the BC6D25, if you are operating in the special double-winding full power mode (by setting the microstep size to 16, or 1 full/step per microstep), you have the option of delivering the full current to each winding.

This means that the minimum current multiplier to use is 2 (i.e., scale your per-winding current by 2), to obtain an approximate maximum current draw per motor. We recommend using a current multiplier of 2.5, to give your supply reserve to account for timing characteristics of the current regulation circuitry.

Obviously, if you are going to run multiple motors off of one supply, you will need to add together all of the currents needed in order to determine how large of a supply to use.

For example, if you are going to operate a motor whose winding current has been calculated to be 0.4 amps, then your power supply needs to be able to supply 2.5 x 0.4, or 1.0 amps to drive that particular motor.

3. Determine the voltage for your motor power supply

Your motor power supply needs to be above that which would provide the base current through the motor, if the supply were directly connected to the motor. From the formula

V=I*R

where "V" is the power supply voltage, "I" is the current, and "R" is the resistance of the motor windings, we can derive a minimum power supply voltage by knowing the current requirement and by measuring the resistance of the windings. For example, if we have a 0.5 amp/winding motor, which has a coil resistance of 10 ohms, then the minimum voltage for the power supply would have to be 0.5 * 10 or 5 volts. The board requires a minimum of a 6.5 volt power supply, therefore you would us at least a 6.5 volt supply for the motor voltage.

Since the controller is a current regulating controller, using a larger supply voltage than the above minimum (while not exceeding the voltage limits for the board) simply increases the top speed of the motor, assuming that you have set the 'H' parameter to the correct value for your motor.

4. Determine the logic supply requirements

The BC6D20 and BC6D25 series require 350-400 mA for operation if the optional fan assembly is installed. They require 80 to 150 mA if no fan is installed.

5. Determine the power supplies you will be using

Your choices are dependent on the desired voltage to the motors, and on the board which you have purchased from us. In all cases, we strongly recommend that linear supplies be used: switching supplies are not very good when used with inductance based loads.

Single Supply.

If your motor power supply voltage is from 6.5 to 15 volts, then you may choose to use a single supply to operate the system. Position the power jumper at the ">6.5" location, and remember to run separate wires for each of the power connector pins to the supply.

Dual or Triple Supply

You may separate the motor supplies from the logic supply. If you do so, you will have the option of providing the 5 volts for the logic system, or of using a 6.5 to 15 volt supply for the logic.

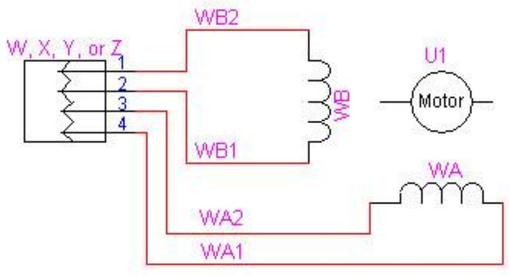
The motor supply should be above 5 volts in all cases (due to some signal requirements on the board), and otherwise is as calculated under sections 1 through 3, above. If the supply is to drive 2 or 4 motors, please remember to double or quadruple the current needs.

ALWAYS PLACE THE POWER JUMPER IN THE CORRECT POSITION TO MATCH YOUR LOGIC SUPPLY VOLTAGE! The jumper must be in the '5V' position if your +Vc supply is 5 volts, and in the ">6.5" position if your +Vc supply is from 6.5 to 15 volts.

Always remember that a "star ground" is required when you operate using any split supply; run all three grounds to a common external point, and run the power supply commons to the same point. This avoids referencing issues in the system.

Wiring Your Motor (BC6D20/BC6D25)

There are six identical connectors used to operate the W, X, Y, Z, A and B motors. The connectors are labeled with respect to which motor they operate. (This designation affects only which commands are to be used to control the motors; no other functionality is changed.) They are wired as follows for the BC6D20 and the BC6D25 series of controllers (pins counting from top to bottom):



Typical Bipolar Motor Connection
To the BC2D15 or BC4D15 Board

Pin	Name	Description
1	WB2	Winding B, pin 2
2	WB1	Winding B, pin 1
3	WA2	Winding A, pin 2
4	WA1	Winding A, pin 1

This pinout was selected to allow simple reversing of the connector (i.e., take it out and turn it around) to reverse the direction of the motor if a non-polarized connector is used.

Stepping sequence, testing your connection

The current is run through these connectors to generate a clockwise sequence as follows:

Step	***WB2***	***WB1***	***WA2***	***WA1***
0	0	0	0	**1**
1	0	**1**	0	**1**
2	0	**1**	0	0
3	0	**1**	**1**	0
4	0	0	**1**	0
5	**1**	0	**1**	0
6	**1**	0	0	0
7	**1**	0	0	**1**

Determining Lead Winding Wire Pairs

If there is no manufacturer’s wiring diagram available, unipolar and bipolar motor windings can both often be identified with an ohm-meter by performing tests of their resistances between the motor leads.

For any motor, number the leads (from 1 to 4 for a bipolar motor, from 1 to 5 or 6 for a unipolar motor). Then measure the resistances and record the values in the empty cells in a table like the following: (IF YOUR MOTOR IS UNIPOLAR, PLEASE CONTACT US. WE CAN USE THE BC6D20 TO OPERATE SOME UNIPOLAR MOTORS IN BIPOLAR MODE!)

	1	**2**	**3**	**4**	**5**	**6**
1	-					
2	-	-				
3	-	-	-			
4	-	-	-	-		
5	-	-	-	-	-	
6	-	-	-	-	-	-

For example, the cell at location (1,2) would be filled in with the resistance between leads 1 and 2. The '-' entries show values which do not need to be separately measured, since they are already measured in another row/column pair (or are a self-reading). For example, having measured the resistance between leads 1 and 2 to fill in cell (1,2), there is no reason to separately measure leads 2 and 1! If you have fewer leads than those shown in the table, ignore the rows and columns with the nonexistent leads.

For a 4-wire bipolar motor, the low-resistance pairs are the opposite ends of matching windings; high-resistance pairs are different windings. For example, if cell (1,2) shows 10 ohms, while (1,3) shows greater than 1000 ohms, then wires 1 and 2 can be called winding A, while wires 3 and 4 can be called winding B.

For a 5-wire unipolar motor, you will observe 2 reading values in the resulting table, with the higher reading being about double that of the lower reading. The single line which has the lower reading on all of its entries in the table is the common lead; the other wires are the winding leads (unfortunately, this test cannot show which is winding A and which is winding B through resistances alone).

For a 6-wire unipolar motor, you will observe 3 reading values in the resulting table.

- If you see a single reading near 0, then the two leads associated with that reading are the common leads, and the remaining 4 wires are the windings WA1, WA2, WB1 and WB2 (this test cannot determine which is winding A or B through resistances alone). As a check, you can observe that all readings between the other wires and either of the 2 common wires have value ½ that of all of the readings between the non-common wires.
- Otherwise, you will see readings which are near infinity (which identify leads from different windings), are at some value (such as 10), or are at double that value (such as 20). The pairs which show the "double value" are the opposite ends of a given winding (i.e., WA1 and WA2, or WB1 and WB2). The remaining wires are the "common" leads for their given windings.

You can often operate a 6-wire unipolar motor as if it were a 4-wire bipolar motor by insulating the common leads and leaving them disconnected. When it works, this usually provides more torque for the motor, but it requires double the voltage (at the same level of current) from the power supply. You cannot operate with this pair of wires disconnected if they are connected together inside the stepper motor -- if the resistance between the common leads is not infinite (less than 100,000 ohms), such a connection exists and you must therefore operate using the regular unipolar wiring scheme utilizing a unipolar motor controller.

Sequence Testing

Always double check all of your power and motor connections before you apply power to the system. If you have reversed any power leads, you will blow out our board and you may blow out your power supply! If you are operating a unipolar motor and you short a common lead to a winding pin (WA or WB), then you will blow out our drivers! Similarly, any winding which is shorted to any other winding may burn out our board. None of these issues are warranted failures; repairs for such are not covered!

After winding lines have been determined, identifying a running sequence can be done by testing the lines using following sequence, connecting to the X motor with clip leads. ***Turn off power*** to the board in between each test, so that power is not on while you change the wiring.

For wires A, B, C, and D (where A, B, C, and D are initially connected to the WA1, WA2, WB1, and WB2 lines) try these orders:

	WA1	***WA2***	***WB1***	***WB2***
1.	A	B	C	D
2.	A	B	D	C
3.	A	D	B	C
4.	A	D	C	B
5.	A	C	D	B
6.	A	C	B	D

Note that in the following discussions, a string of characters is shown as being typed by you (for example, `1000G`). Any of our boards will normally send a newline character each time they see a letter (such as the 'G' in the above sequence), and they will echo an asterisk (`*`) when they have completed processing the command. Therefore, your commands will all get an immediate response from the controller; **always wait for the '*' before you continue typing.**

Also, **all commands should be treated as case sensitive** (i.e., 't' may mean something different from 'T'). Please use the case of the command as shown in the following lists.

For each pattern, request a motor motion in each direction using the rules:

Determine the correct value for the 'H' command to use to set the target current for your motor, then issue it as needed.

For example, on the BC6D20 firmwares, 'H' units are normally in units of milliamps. If your motor is a 0.5 amp/winding motor, you would therefore issue the command:

- `Send:    500H`
- `Receive: *   (i.e., wait for board to echo '*' back to you)`

Issue the command sequence

- `Send:    1000X`
- `Receive: *   (i.e., wait for board to echo '*' back to you)`
- `Send:    G`
- `Receive: *`
- `Send:    I`
- `Receive: * (this may take a while; it will be sent by the board in response to the above 'I' when the motor stops)`
- `Send:    0X`
- `Receive: *`
- `Send:    G`
- `Receive: *`
- `Send:    I`
- `Receive: *`

which should cause the motor to spin to logical location 1000, then return to location 0. Wait for the `*` response after each sub-command (the "`X`", "G", and "I" commands) before typing the next character, in order to allow the firmware to finish processing the request.

Only when the motor is wired correctly will you get smooth motion first in one direction and then the other.

Once a possible pattern has been determined, you may find that the direction of rotation is reversed from that desired. To reverse the rotation direction, you can either turn the connector around (this may be the easiest method, if a SIP style connector is used), or you can swap both the WA (swap pin 2 with pin 3) and WB pins (swap pin 4 with pin 5). For example, to reverse

A B C D, rewire as

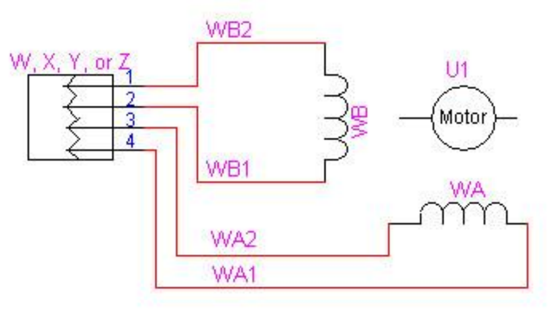
B A D C.

Motor Wiring Examples (BC6D20/BC6D25)

The systems have been tested with an interesting mix of stepper motors, both unipolar and bipolar. All were purchased from Jameco (www.jameco.com). The following sections summarize some of the motors tested.

Bipolar Motors

This section shows some bipolar motors which were used. They only work on the BiStep products. In each case, the wiring is:



Typical Bipolar Motor Connection
To the BC2D15 or BC4D15 Board

Jameco 117954 5 Volt, 0.8 Amp, 7.5 deg/step

This unit is an Airpax LB82773-M1 2 phase bipolar stepping motor. This motor does NOT microstep at all. It may only be used in full and half step modes (i.e. use microstep sizes of 8 and 16)! Other sizes may be used, but that motor will not really stop at any other than ½ step locations.

The wiring of this unit is:

Color	**BC6D20**
Yellow	WB2
Black	WB1
Red	WA2
Gray	WA1

The “H” command to be issued after any power-on or reset action in order to select the correct current for motor operation is:

800H

Jameco 155459 12 Volt, 0.4 Amp, 2100 g-cm, 1.8 deg/step

This unit is a GBM 42BYG023 stepping motor, which provides for 2100 g-cm of holding torque. It may be wired as:

Color	**BC6D20**
Brown	WB2
Orange	WB1
Yellow	WA2
Red	WA1

The "H" command to be issued after any power-on or reset action in order to select the correct current for motor operation is:

400H

Jameco 163395 8.4 Volt, 0.28 Amp, 0.9 deg/step

This is a Scotts Valley 5017-935 stepper motor. It may be wired as:

Color	**BC6D20**
Yellow	WB2
White	WB1
Blue	WA2
Red	WA1

The "H" command to be issued after any power-on or reset action in order to select the correct current for motor operation is:

280H

Jameco 168831 12 Volt, 1.25 Amp

This motor is a Superior Electric "SLO-SYN" stepping motor, model number SM-200-0050-HL. We tested it with the wiring of:

Color	**BC6D20**
White/Brown	WB2
Brown	WB1
White/Yellow	WA2
Yellow	WA1

The "H" command to be issued after any power-on or reset action in order to select the correct current for motor operation is:

```
1250H
```

Install The Fan Assembly (BC6D20/BC6D25)

When the BC6D20 or BC6D25 unit is purchased, a fan assembly may optionally be included in order to provide cooling for the unit. The fan assembly is shipped separately; it is not mounted onto the board during shipment.

The fan assembly is mounted onto the BC6D20/BC6D25 via four 1-inch stand-offs, and is powered off of a 2-pin SIP header that is available on the board.

Please review the figure 1 on the following page which shows that 2 pin connector near the left center of the board.

1. Attach the four 1-inch stand-offs to the BC6D20/BC6D25. Use the four nylon washers on the bottom of the board and the four nuts to attach the stand-offs; they are positioned as shown in figure 1 on the following page labeled "Unit with Mounting Posts". Please be careful to not short any components with the mounting post. with the non-conductive washer mounted between the board and the nut, there is no chance of shorting anything with the nut on the bottom of the board.
2. Attach the mounting bracket to the fans. See figure 2 for the parts, and figure 5 to see what it should look like when assembled.
    a. Position the 'label' side of the fans **DOWN**
    b. position the power plug to the left
    c. rotate the right fan so that its power would come out in between the fans
    d. mount the PCB bracket such that the two fans are connected to each other as best shown in figure 5. Route the wire that goes to the power connector in between the screw and the fan body.
3. Connect the fan power to the 2 pin "Fan" header at the left side of the board (this will be under the fans once they are installed). The black lead goes to the bottom pin the red lead goes to the top pin on the header (by the silkscreen that says "+FAN"). See the figures 3 and 4 on the next page.
4. Attach the fans to the stand-offs using the remaining screws and lockwashers. **The label side of the fans must face down** as shown in the photo on the next page, labeled "Unit with Fans". Route the fan power wire between the screw and the fan body as shown on figure 4.

It is absolutely critical that the direction of air flow be towards the board; otherwise, the board will fail! Any failure induced by incorrect mounting of the fan assembly is not covered by the warranty.

You should (hopefully) now have an operational board. Apply power, and observe whether the LED illuminates, ***and the fans blow air towards the surface of the board***. If either action does not occur, turn the system off, and trace the problem.

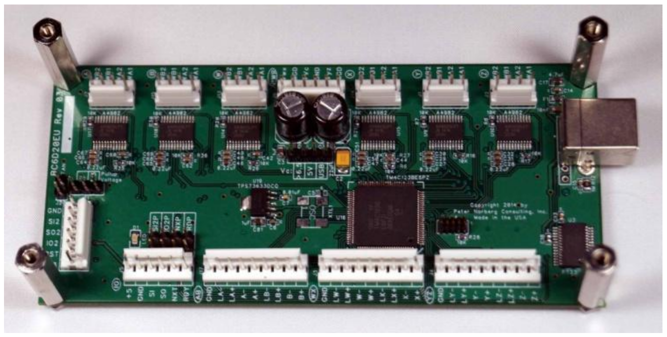


Figure 1: BC6D20 Unit with mounting posts

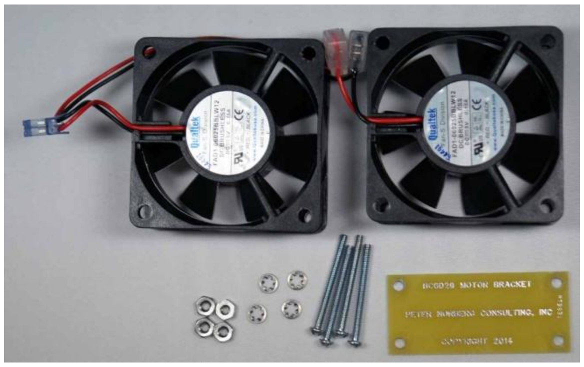


Figure 2: Fans and mounting bracket

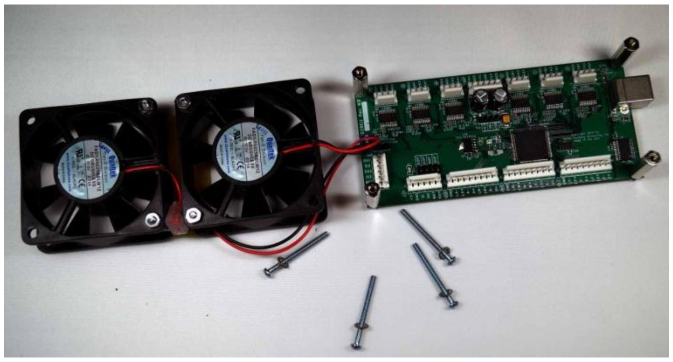


Figure 3: Unit with fans attached to fan power – Red wire on 'top/left'

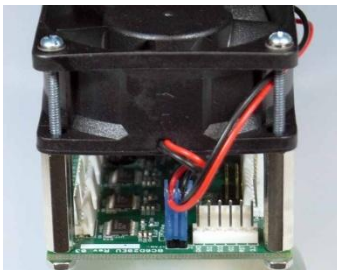


Figure 4: Fans Mounted showing fan power plug and routing of wire

Please note that the wires should be routed between the fan and the mounting screw.

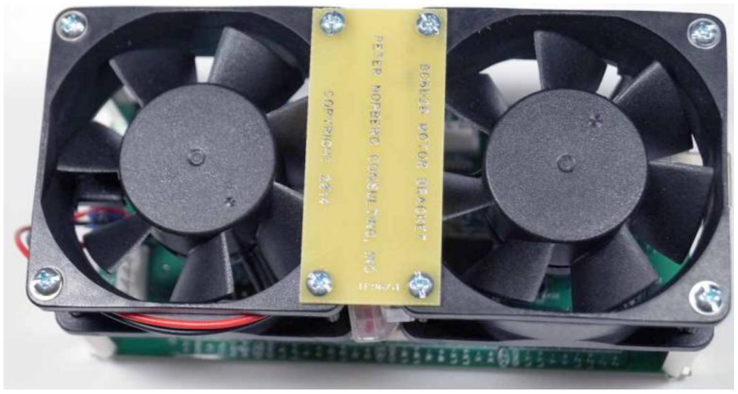


Figure 5: Unit with fans fully installed