Qik 2s9v1 User's Guide

1. Overview

The qik 2s9v1 is Pololu's second-generation dual serial motor controller. The compact board allows any microcontroller or computer with a serial port (external RS-232 level converter required) to drive two small, brushed DC motors with full direction and speed control. The improvements over the previous generation and competing products include:

- high-frequency PWM to eliminate switching-induced motor shaft hum or whine
- a robust, high-speed communication protocol with user-configurable error condition response
- visible LEDs and a demo mode to help troubleshoot problematic installations
- reverse power protection on the motor supply (not on the logic supply)

Pololu qik 2s9v1 dual serial motor controller.

Main features of the Qik 2s9v1

- Simple bidirectional control of two DC brush motors.
- 4.5 V to 13.5 V motor supply range.
- 1 A maximum continuous current per motor.
- 2.7 V to 5.5 V logic supply range.
- Logic-level, non-inverted, two-way serial control for easy connection to microcontrollers or robot controllers.
- Optional automatic baud rate detection.
- Two on-board indicator LEDs (status/heartbeat and serial error indicator) for debugging and feedback.
- Serial error output to make it easier for the main controller to recover from a serial error condition.
- Jumper-enabled demo mode allowing initial testing without any programming.

- Optional CRC error detection eliminates serial errors caused by noise or software faults.
- Optional motor shutdown on serial error or timeout for additional safety.

Specifications

Motor channels:	2
Motor supply voltage:	4.5 – 13.5 V
Continuous output current per channel:	1 A
Peak output current per channel:	3 A
Auto-detect baud rate range:	1200 – 38400 bps
Fixed baud rate:	38400 bps
Available PWM frequencies:	31.5 kHz, 15.7 kHz, 7.8 kHz, 3.9 kHz
Logic supply voltage:	2.7 – 5.5 V
Reverse voltage protection?:	Y (on motor supply only)
Motor driver:	TB6612FNG

Important safety warning

This product is not intended for young children! Younger users should use this product only under adult supervision. By using this product, you agree not to hold Pololu liable for any injury or damage related to the use or to the performance of this product. This product is not designed for, and should not be used in, applications where the malfunction of the product could cause injury or damage. Please take note of these additional precautions:

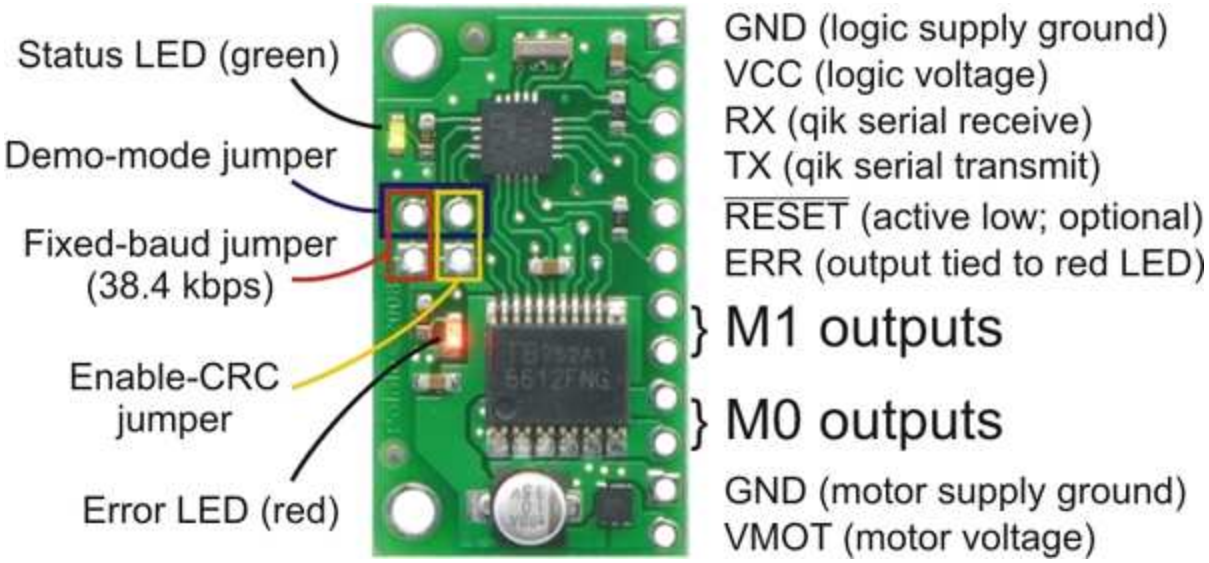
- This product contains lead, so follow appropriate handling procedures, such as not licking the product and washing hands after handling.
- Since the PCB and its components are exposed, take standard precautions to protect this product from ESD (electrostatic discharge), which could damage the on-board electronics. When handing this product to another person, first touch their hand with your hand to equalize any charge imbalance between you so that you don't discharge through the electronics as the exchange is made.
- Review the instructions carefully before making any electrical connections, and do all wiring while the power is turned off. Incorrect or reversed wiring could cause an electrical short or unpredictable behavior that damages this product and the devices it is connected to.
- This product is designed to be connected to motors, which should be operated safely. Wear safety glasses, gloves, or other protective equipment as appropriate, and avoid dangerous situations such as motors spinning out of control by designing appropriate safeguards and limits into your projects.

2. Contacting Pololu

You can check the **qik 2s9v1 dual serial motor controller page** for additional information. We would be delighted to hear from you about any of your projects and about your experience with the qik motor controller. You can **contact us** directly or post on our **forum**. Tell us what we did well, what we could improve, what you would like to see in the future, or anything else you would like to say!

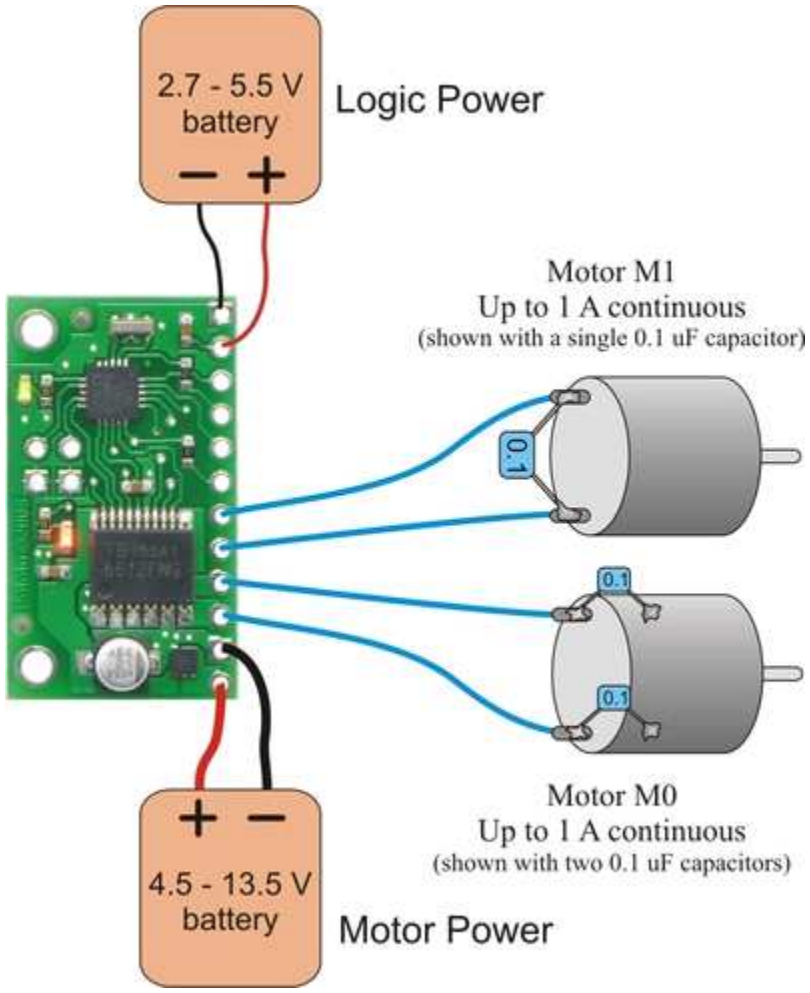
3. Connecting the Qik

Connecting to the qik can be as simple as hooking up logic and motor power, your motors, and RX. Many applications can ignore the jumpers and leave the TX, ERR, and RESET pins disconnected.



The qik connections are shown above, and most of the pins are labeled on the back side of the motor controller. All square pads are ground.

3.a. Power and Motors



The qik motor controller takes two power inputs: motor power supplied via the **VMOT** and **GND** pins at the bottom of the board and logic power supplied via the **VCC** and **GND** pins at the top of the board.

Motor Power

The qik is designed to independently drive up to two bidirectional brushed DC motors, referred to as M0 and M1. The two terminals of each motor should connect qik as shown above. Variable speed is achieved with 7-bit or 8-bit pulse width modulated (PWM) outputs at a frequency of 31.5 kHz, 15.7 kHz, 7.8 kHz, or 3.8 kHz. The resolution and frequency can be set via the qik's PWM configuration parameter (see **Section 5.c**). The motor

direction convention used in this document is that "forward" corresponds to grounding the – pin while PWMing the + pin between V_{MOT} and ground. "Reverse" corresponds to grounding the + pin while PWMing the – pin between V_{MOT} and ground. See **Section 5.e** for more information on the motor commands.

The motor voltage must be between 4.5 and 13.5 V, and your motor power source must be capable of supplying the current your motors will draw. The qik 2s9v1 motor controller uses the TB6612FNG dual motor driver, which is capable of supplying a continuous 1 A per motor channel, with a peak output of 3 A per channel. Continuous performance is a function of how well you can keep the motor driver cool; addition of a heat sink or increased air flow over the board can allow the driver to output higher currents without overheating. The driver has thermal protection that causes it to shut down when it detects that it is overheating, but it is not a good idea to rely upon this thermal protection to prevent damage to the unit that can be caused by using it out of spec.

Logic Power
The logic voltage must be between 2.7 and 5.5 V. Voltages that fall below 2.7 V will trigger a brown-out and cause the device to reset. **Note that the voltage on the qik's serial input must not exceed Vcc.**

While it is possible to use the same power source for both your motors and logic, we recommend against doing this. Motors will typically introduce a lot of noise to the motor power rail. If this same power is used for logic, the noise will negatively impact the functioning of the device. If you want to use the same power source for both motors and logic, you should decouple the two by using large capacitors and/or a regulator. Additionally, even if you use separate power supplies for your motors and logic, you should take whatever steps you can to limit electrical noise from your motors. For example, solder 0.1 uF capacitors across your motor terminals (as shown in the diagram above); using three capacitors per motor (one across the terminals and one from each terminal to the motor can) results in maximum noise suppression. You can further decrease motor noise by keeping your motor leads as short as possible and twisting the motor leads around each other in a helix

> **Note:** Even if your motor and logic power sources share a common ground, make sure you connect your motor power supply's ground to the GND pin just above the VMOT pin. This ground pin is designed to handle the higher currents that your motors will draw; the ground pin above the VCC pin is not intended for high currents.

3.b. Control Signals and Motors

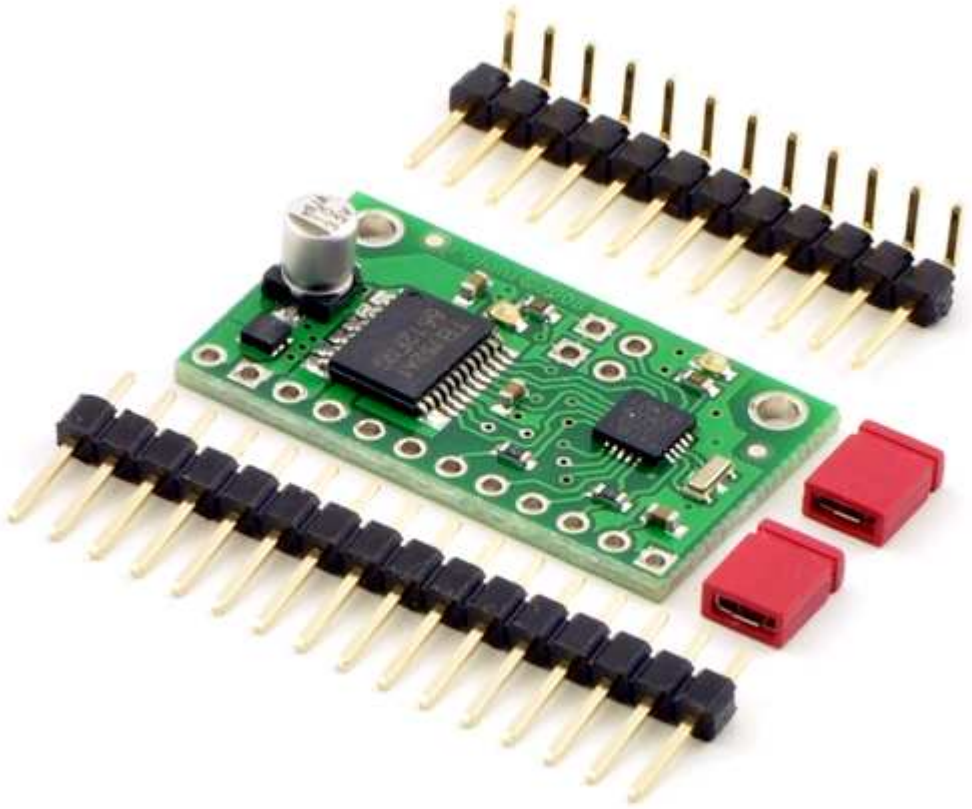
Serial Lines
The qik requires a logic-level (0 – Vcc), non-inverted serial input connected to its serial receive line **RX**. This type of serial is often referred to as TTL. The voltage on this pin should not exceed Vcc. The qik will provide logic-level serial output on its serial transmit line **TX** in response to commands that request information. Information requests always result in the transmission of a single byte per request. If you aren't interested in receiving feedback from the qik, you can leave this line disconnected.

Note that these lines are not compatible with RS-232 serial, which is inverted and uses voltages that would be out of spec (e.g. -12V to 12V). To connect the qik to an RS-232 device, you will need to use a converter such as our **23201a serial adapter**, or a level shifter and inverter.

$\overline{\text{Reset}}$
The reset line **RST** is an active-low input, which means that it will reset the qik when driven low. This pin is internally pulled high, so many applications can leave this pin disconnected.

Error
The error line **ERR** is an output that is connected to the red error LED and will drive high in response to an error (which in turn lights the LED). Once an error occurs, the pin outputs high until a serial command is issued to read the error byte, at which point the pin goes to a high-impedance state that is pulled low through the LED. This allows you to connect the error lines of multiple qiks to the same digital input. For more information on this, please see **Section 5.b**. If you don't care about error detection, you can leave this pin disconnected.

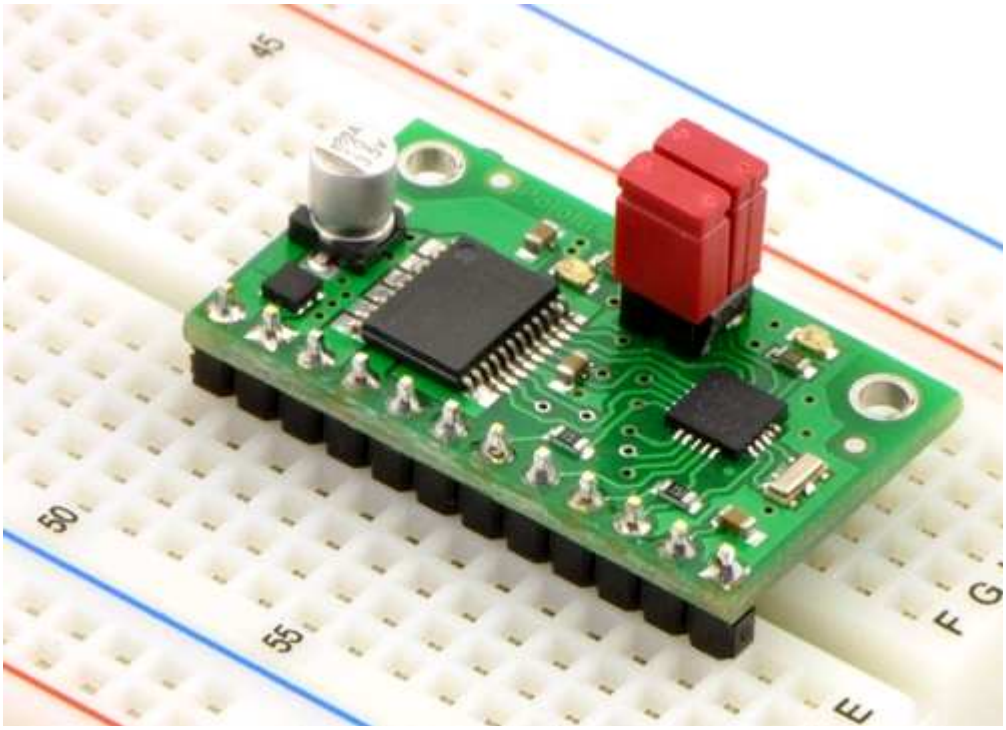
3.c. Included Hardware



The qik ships with a 16×1 **straight 0.100" male header strip**, a 12×1 **right angle 0.100" male header strip**, and two **red shorting blocks**. This hardware gives you several options when it comes to making connections to your qik.

For the most compact installation, you can solder wires directly to the qik pins themselves and avoid using any of the included hardware at all.

If you want to make less permanent connections, you can break the header strip of your choice into a 12×1 piece and two 2×1 pieces and solder these strips into the qik control pins and jumper pins respectively. You can then make your own cables that have **female headers** on them and plug these onto the male headers on your qik, or you can simply plug your qik into a breadboard. Using the right angle header allows for a compact profile or for vertical mounting into a breadboard, while using the straight header allows for breadboarding as shown in the picture below.



3.d. Jumpers
The qik jumpers allow you to alter the behavior of the device if you are not happy with its default behavior. These jumpers can be left off for most applications. If you use a jumper, it must be in place when the unit first starts up; changing the jumpers while the unit is running will have no effect.

Fixed-Baud Mode
The jumper labeled **A** on the bottom of the qik (i.e. the one closest to the edge of the board) will set the qik to fixed-baud mode when the shorting block is in place. The default behavior of the qik is auto-detect mode, where the qik determines the baud rate automatically when it receives the first 0xAA (170) byte. If you have a noisy serial connection or find that the automatic baud detection is not working well for your application, you can use a shorting block or some other jumper to ground pin A (the circular pin right next to the "A" label). Doing this will fix the qik's baud rate at 38,400 bps. In fixed-baud mode, the qik will skip the automatic baud detection phase that normally occurs on start-up.

Enable-CRC Mode
The jumper labeled **B** on the bottom of the qik will enable cyclic redundancy check (CRC) mode when the shorting block is in place. This allows you to increase the robustness of your qik connection through the addition of a CRC error-checking byte to the end of the command packets you send to the qik. The default behavior of the qik is to simply respond to a command packet once it receives the last byte. Shorting pin B (the circular pad right next to the "B" label) to ground will cause the qik to expect an additional byte at the end of the command byte that results from a CRC-7 computation on the entire message. If this byte does not match the expected CRC, the qik will ignore the command and will use the ERR pin to announce a CRC error. Please see **Section 6** for more information on how CRC error detection works.

Demo Mode
If you short pin A to pin B and reset the qik, you will enter demo mode and remain in demo mode for as long as the short is maintained. Demo mode gives you an easy way to test your qik and troubleshoot your application for potential problems. In demo mode, the qik will smoothly ramp motor 0 from stopped to full-speed forward to full-speed reverse to stopped again over a few seconds. It will then do the same for motor 1. While motor 0 is active, the red LED is on; while motor 1 is active, the red LED is off. While a motor is being driven forward, the green LED is on; while a motor is being driven in reverse, the green LED is off.

Demo mode can help you determine before you've even written any code if you have an issue with your power supply, such as insufficient ability to supply the current your motors are drawing or interference from motor noise.

While in demo mode, any serial data that is received by the qik on the RX line will be echoed on the TX line. This gives you an easy way to test your serial connection. Baud rates should be 38,400 bps or less while in demo mode.

3.e. Indicator LEDs
The green LED serves as a general heartbeat indicator that lets you know your qik is alive and what state it's in.

Automatic Baud Detection Phase
When the qik first starts up in automatic baud detection mode, it enters a phase in which it is waiting to receive the byte 0xAA at a baud rate that is within the range of 1200 bps and 38,400 bps. If the serial receive line is pulled high, as is expected for an idle TTL serial line, the green LED will flash on and off evenly several times a second. Specifically, it will cycle between being on for 200 ms and off for 200 ms. If the serial receive line is low, this is indicative of a bad serial connection and the red (error) LED will cycle between being on for 200 ms and off for 200 ms (and the green

LED will be off).

If a serial byte other than 0xAA is received in this mode, or if 0xAA is transmitted at an invalid baud rate, the red (error) LED will turn on and stay on until the automatic baud detection phase ends. This gives you feedback that the baud has not yet successfully been set and you are still in the automatic detection phase. Once the baud is detected, this phase ends and the qik proceeds to normal operation.

Normal Operation
In normal operation, the green heartbeat LED will very briefly flash once every two seconds. If any serial activity is detected, the green LED turns on until the next heartbeat turns it off. If a serial error occurs, the red LED turns on and remains on until you issue a get-error command (see **Section 5.b**).

Demo Mode
In demo mode, the LEDs will cycle through the pattern:

1. red and green on
2. red on and green off
3. red off and green on
4. red and green off

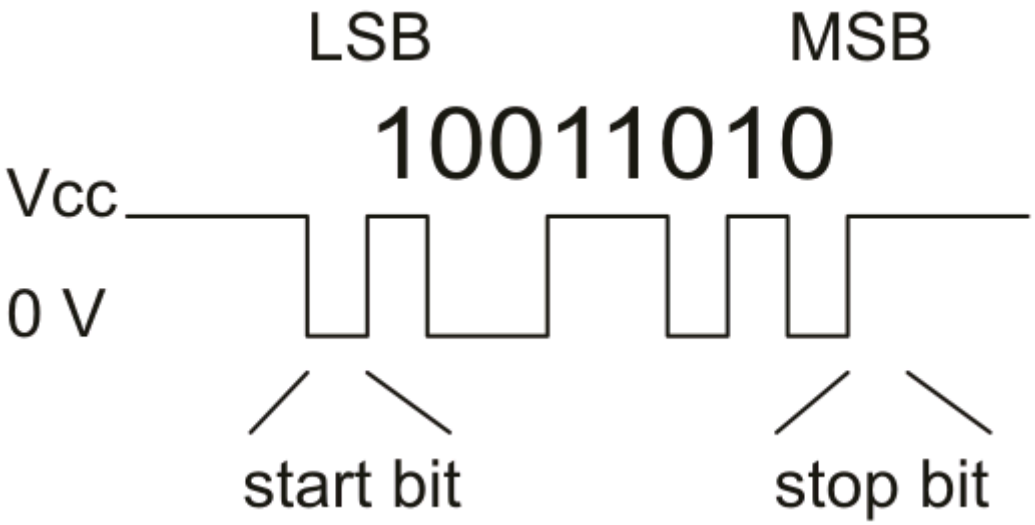
This cycle will take a few seconds and will correspond to the outputs of the qik's two motor ports.

4. Serial Interface

You can use the serial interface for three general purposes: querying the qik for information, setting its configuration parameters, and sending it motor commands. Motor commands are strictly one-way; all other commands result in the qik's responding with a single byte that either represents information that has been requested or feedback on the effect of the issued command.

4.a. TTL Serial

The qik requires a logic-level (0 – Vcc, or "TTL"), non-inverted serial input connected to its serial receive line **RX**. The serial interface is *asynchronous*, meaning that the sender and receiver each independently time the serial bits; asynchronous serial is available in computer serial ports (though at non-TTL levels) and as hardware modules called "UARTs" on many microcontrollers. Asynchronous serial output can also be "bit-banged" by a standard digital output line under software control. The data format is 8 data bits, one stop bit, with no parity. The diagram below depicts a typical asynchronous, non-inverted TTL serial byte.



A non-inverted TTL serial line has a default (non-active) state of high. A transmitted byte begins with a single low "start bit", followed by the bits of the byte, least-significant bit (LSB) first. Logical ones are transmitted as high (Vcc) and logical zeros are transmitted as low (0 V), which is why this format is referred to as "non-inverted" serial. The byte is terminated by a "stop bit", which is the line going high for at least one bit time. Because each byte also requires start and stop bits, each byte takes 10 bit times to transmit, so the fastest possible data rate in bytes per second is the baud rate divided by ten. At the maximum baud rate of 38,400 bits per second, the maximum realizable data rate, with a start bit coming immediately after the preceding byte's stop bit, is 3,840 bytes per second.

The voltage on the RX pin should not exceed Vcc. The qik will provide logic-level serial output on its serial transmit line **TX** in response to commands that request information. Information requests aways result in the transmission of a single byte per request. If you aren't interested in receiving feedback from the qik, you can leave this line disconnected.

> **Note:** These lines are not compatible with RS-232 serial, which is inverted and uses voltages that would be out of spec (e.g. -12V to 12V). To connect the qik to an RS-232 device, you will need to use a converter such as our **23201a serial adapter**, or a level shifter and inverter.

4.b. Baud Rates
The qik can handle baud rates between 1200 and 38,400 bps. In its default state, the qik will start up in an automatic baud detection phase, where it waits for you to send it the byte 0xAA (170). The qik will detect the baud rate you are using from this byte and proceed to the normal operation phase. If you have the fixed-baud jumper in place, the qik will skip the autodetect phase and will instead immediately begin normal operation at a baud rate of 38,400 bps. Please see **Section 3.d** for more information.

4.c. Command Protocols
Once the qik has entered the normal operation phase, you can control it by issuing serial commands. If your qik is set to automatically detect the baud, you must first send it the byte **0xAA** (170) in order to exit autodetect mode and enter normal operation.

The qik serial command protocol is fairly straightforward. Communication is achieved by sending command packets consisting of a single command byte followed by any data bytes that command requires. Command bytes always have their most significant bits set (i.e. range from 128 – 255) while data bytes always have their most significant bits cleared (i.e. range from 0 – 127). This means that each data byte can only transmit seven bits of information.

One significant improvement over earlier Pololu serial controllers is the qik's error handling, which allows the user to specify responses to serial errors (which include bad commands, incorrectly formatted commands, or even hardware-level serial errors). The qik has a configuration parameter that, if set, will shut down the motors if a serial error occurs, but the qik itself will continue running and accepting commands. This is a safety precaution taken in case the serial error occurred during a command that was intended to stop the motors.

The qik responds to two sub-protocols:

Compact Protocol:
This is the simpler and more compact of the two protocols; it is the protocol you should use if your qik is the only device connected to your serial line. The qik compact protocol command packet is simply:

> **command byte (with MSB set), any necessary data bytes**

For example, if we want to set motor M1 to full speed forward using the compact protocol, we could send the following byte sequence:

in hex: **0x8D, 0x7F**
in decimal: **141, 127**

The byte 0x8D is a command for M1 forward, and the data byte contains the motor speed. Note that every qik command byte will start with **0x8_** when using the compact protocol.

Pololu Protocol:
This protocol is compatible with the serial protocol used by our other serial motor and servo controllers. As such, you can daisy-chain a qik on a single serial line along with our other serial controllers (including additional qiks) and, using this protocol, send commands specifically to the desired qik without confusing the other devices on the line.

The Pololu protocol is to transmit 0xAA (170 in decimal) as the first (command) byte, followed by a device-number data byte. The default device number for the qik is **9**, but this is a configuration parameter you can change. Any **qik** on the line whose device number matches the specified device number will accept the command that follows; all other Pololu devices will ignore the command. The remaining bytes in the command packet are the same as the compact protocol command packet you would send, with one key difference: the compact protocol command byte is now a data byte for the command 0xAA and hence must have its most significant bit cleared. Therefore, the command packet is:

> **0xAA, device # byte, command byte with MSB cleared, any necessary data bytes**

For example, if we want to set motor M1 to full speed forward for a qik with device number 9, we could send the following byte sequence:

in hex: **0xAA, 0x09, 0x0D, 0x7F**
in decimal: **170, 9, 13, 127**

Note that 0x0D is the command 0x8D with its most significant bit cleared. Since all compact-protocol command bytes start with **0x8*n***, these bytes will all turn into data bytes **0x0*n***.

The qik will respond to both the Pololu and Compact protocols on the fly; you do not need to use a jumper or configuration parameter to identify

which protocol you will be using.

Procedure for Daisy-Chaining
Daisy-chaining multiple qiks together on the same serial line is simple. Individually assign each qik a different device ID using the set configuration parameter command (see 0J303), and then connect your TTL serial transmit line to each qik's RX line. If you wish, you can connect all of the qiks' ERR lines to a single input on your controlling module. When you issue your first Pololu-protocol command, the qiks will all automatically detect the baud from the initial 0xAA byte.

Connecting multiple serial outputs to one serial input is more complicated. Each device will only transmit when requested, so if each unit is addressed separately, multiple units will not transmit simultaneously. However, the outputs are driven, so they cannot simply be wired together. Instead, you can use an AND gate (since the idle state is high).

If you want to daisy-chain a qik with other Pololu devices that use 0x80 as an initial command byte, the procedure becomes slightly more complicated. You should first transmit the byte 0x80 so that these devices can automatically detect the baud rate, and only then should you send the byte 0xAA so that the qik can detect the baud rate. Once all devices have detected the baud rate, Pololu devices that expect a leading command byte of 0x80 should ignore command packets that start with 0xAA and qiks will ignore command packets that start with 0x80.

5. Serial Commands

5.a. 0x81: Get Firmware Version
Compact protocol: **0x81**
Pololu protocol: **0xAA, *device ID*, 0x01**

This command returns the a single ASCII byte that represents the version of the firmware running on the qik. All qiks produced so far have firmware version '1'.

5.b. 0x82: Get Error Byte
Compact protocol: **0x82**
Pololu protocol: **0xAA, *device ID*, 0x02**

The qik maintains an error byte, the bits of which, when set, reflect various errors that have been detected since the byte was last read using this command. If we call the least-significant bit 0, the bits of the error byte are as follows:

- **bits 0 – 2:** unused
- **bit 3:** Data Overrun Error
  A hardware-level error that occurs when serial data is received while the hardware serial receive buffer is full. This error should not occur during normal operation.
- **bit 4:** Frame Error
  A hardware-level error that occurs when a byte's stop bit is not detected at the expected place. This error can occur if you are communicating at a baud rate that differs from the qik's baud rate.
- **bit 5:** CRC Error
  This error occurs when the qik is running in CRC-enabled mode (i.e. the CRC-enable jumper is in place) and the cyclic redundancy check (CRC) byte added to the end of the command packet does not match what the qik has computed as that packet's CRC. In such a case, the qik will ignore the command packet and generate a CRC error. See **Section 6** for more information on cyclic redundancy checking.
- **bit 6:** Format Error
  This error occurs when the qik receives an incorrectly formatted or nonsensical command packet. For example, if the command byte does not match a known command, data bytes are outside of the allowed range for their particular command, or an unfinished command packet is interrupted by another command packet, a format error will be generated.
- **bit 7:** Timeout
  It is possible to use a configuration parameter to enable the qik's serial timeout feature (see **Section 5.c**). When enabled, the qik will generate a timeout error if the timeout period set by the configuration parameter elapses. The timeout timer is reset every time a valid command packet is received.

An error will cause the red LED to light and the ERR pin to drive high until this command is called. Calling this command will clear the error byte, turn the red LED off, and set the ERR pin as high-impedance (causing it to be pulled low through the LED). If *shutdown-on-error* configuration parameter is set to 1, motors M0 and M1 will be stopped as a safety precaution when any of these errors occurs (see **Section 5.c**).

5.c. 0x83 & 0x84: Get & Set Configuration Parameter
The qik has four configuration parameters that are saved in non-volatile memory, and uses commands 0x83 and 0x84 to read and write these parameter values, respectively. The parameters are numbered as follows:

Configuration Parameters
- **0:** Device ID

This parameter determines which device ID the unit will respond to when the Pololu protocol is used. It has a default value of 9 (0x09 in hex) and can be set to any value from 0 – 127. When setting this parameter, you should only have one qik on your serial line at a time.

- **1:** PWM Parameter
  This parameter determines frequency and resolution of the pulse width modulation (PWM) signal used to control motor speed. Note that setting this parameter while the motors are running causes them to stop.

  The least significant bit (bit 0) selects for 7-bit resolution when cleared (i.e. full motor speed is 127) and 8-bit resolution when set (i.e. full motor speed is 255). A PWM with 7-bit resolution has twice the frequency of one with 8-bit resolution.

  Bit 1 of this parameter selects for high-frequency mode when cleared and low-frequency mode when set. Using high-frequency mode puts the PWM frequency outside the range of human hearing if you are also in 7-bit mode (or very close to it if you are in 8-bit mode), which can help you decrease motor noise. Using low frequency mode has the benefit of decreasing power losses due to switching.

  The default value for this parameter is 0 (high-frequency 7-bit mode, resulting in a PWM frequency of 31.5 kHz).

  Valid values for this parameter are:
  - 0 = high-frequency, 7-bit mode (PWM frequency of **31.5 kHz**, which is ultrasonic)
  - 1 = high-frequency, 8-bit mode (PWM frequency of **15.7 kHz**)
  - 2 = low-frequency, 7-bit mode (PWM frequency of **7.8 kHz**)
  - 3 = low-frequency, 8-bit mode (PWM frequency of **3.9 kHz**)

- **2:** Shutdown Motors on Error
  When this parameter has a value of 1, both motors M0 and M1 are stopped as a safety precaution whenever an error occurs; otherwise, if this parameter has a value of 0, errors will not affect the motors. For more information on the various types of errors that can occur, see **Section 5.b**. This parameter has a default value of 1 (shut down the motors on any error) and valid values for this parameter are 0 or 1.

- **3:** Serial Timeout
  When this parameter has a value of 0, the serial timeout feature is inactive. Otherwise, the value of this parameter controls how much time can elapse between receptions of valid command packets before a serial timeout error is generated. This can be used as a general safety feature to allow the qik to identify when communication with the controlling device is lost and shut down the motors as a result (assuming the *shutdown motors on error* parameter set to a value of 1).

  The timeout duration is specified in increments of 262 ms (approximately a quarter of a second) and is calculated as the lower four bits (which are interpreted as a number from 0 – 15) times two to the upper three bits (which are interpreted as a number from 0 – 7). If the lower four bits are called **x** and the upper three bits are called **y**, the equation for the length of the timeout duration would be:

  timeout = 0.262 seconds * x * 2^y

  For example, if the timeout parameter is set as 0x5E (01011110 in binary), we have that **x** = 1110 (binary) = 14 (decimal) and **y** = 101 (binary) = 5 (decimal), which results in a timeout duration of

  0.262s * 14 * 2^5 = **117 seconds**.

  The maximum timeout duration (arising from a parameter value of 0x7F, or 127 in decimal) is 8.32 minutes and the minimum timeout duration (arising from a parameter value of 1) is 262 ms.

  This parameter has a default value of 0 (serial timeout disabled) and can be set to any value from 0 – 127.

Command 0x83: Get Configuration Parameter
Compact protocol: **0x83,** ***parameter number***
Pololu protocol: **0xAA,** ***device ID*****, 0x03,** ***parameter number***

This command lets you request the current value of any of the four configuration parameters detailed above. This command will cause the qik to transmit a single byte that represents the requested parameter value. If you request an invalid parameter (i.e. if *parameter number* ≥ 4), the value transmitted by the qik should be 0xFF (255 in decimal) and a format error will be generated.

Command 0x84: Set Configuration Parameter
Compact protocol: **0x84,** ***parameter number*****,** ***parameter value*****, 0x55, 0x2A**
Pololu protocol: **0xAA,** ***device ID*****, 0x04,** ***parameter number*****,** ***parameter value*****, 0x55, 0x2A**

This command lets you set the value of any of the four configuration parameters detailed above. The final two bytes of the command packets are format bytes that make it more difficult for this command to be unintentionally or accidentally sent, as might result from a noisy serial connection or buggy code. If either of the format bytes differs from the expected value, the command is ignored and a format error is generated.

It takes the qik approximately 4 ms to finish processing this command, at which point the qik will transmit a single return byte that contains information about whether the process was successful. You should not send commands to the qik until you have received this return byte, or until at least 4 ms have elapsed. The return byte can have the following values:

- **0:** Command OK (success)
- **1:** Bad Parameter (failure due to invalid parameter number)
- **2:** Bad value (failure due to invalid parameter value for the specified parameter number)

Failure will result in a format error.

Once you have set the value of a configuration parameter, that value will be saved even if the unit is unplugged or reset.

5.d. 0x86 - 0x87: Motor Coast
Command 0x86: Motor M0 Coast
Compact protocol: **0x86**
Pololu protocol: **0xAA, *device ID*, 0x06**

Command 0x87: Motor M1 Coast
Compact protocol: **0x87**
Pololu protocol: **0xAA, *device ID*, 0x07**

These commands will set the specified motor to coast by setting the motor outputs to high impedance, which lets the motor turn freely. This is in contrast to setting the motor speed to 0, which sets the motor outputs to ground and acts as a brake, resisting motor rotation.

You can see the effect of this yourself just by using your motor. First, with your motor disconnected from anything, try rotating the output shaft and note how easily it turns. Then, short the two motor leads together and try rotating the output shaft again. You should notice significantly more resistance while the leads are shorted together. The motor-coast command is equivalent to disconnecting your motor and letting it turn freely while setting the motor speed to zero is equivalent to shorting the motor leads together.

5.e. 0x88 - 0x8F: Set Motor Forward/Reverse
The qik can independently control two bidirectional brushed DC motors, driving each either forward or reverse with either 7- or 8-bit speed resolution. In 7-bit mode, motor speed ranges from stopped to full speed as the speed parameter ranges from 0 to 127. In 8-bit mode, motor speed ranges from stopped to full speed as the speed parameter ranges from 0 – 255. The speed resolution can be controlled by the PWM configuration parameter (see **Section 5.c**).

The motor direction convention used in this document is that "forward" corresponds to grounding the – pin while PWMing the + pin between V_{MOT} and ground. "Reverse" corresponds to grounding the + pin while PWMing the – pin between V_{MOT} and ground. Notions of "forward" and "reverse" are somewhat arbitrary. See **Section 3.a** for information about motor and power connections.

Motor M0 Commands
Commands 0x88 – 0x8B apply to motor M0. In 8-bit mode, commands 0x89 and 0x8B will set the motor speed to 128+*motor speed*; in 7-bit mode, command 0x89 is identical to command 0x88 and command 0x8B is identical to command 0x8A.

For example, in 8-bit mode, the command packet **0x88, 0x7F** will set motor M0 speed to 127 out of a maximum of 255, which will result in the motor's turning at half speed. The command packet **0x89, 0x7F** will set motor M0 speed to 127+128 = 255, which will result in the motor's turning at full speed. In 7-bit mode, both commands will set motor M0 speed to 127 out of a maximum of 127, which will result in the motor's turning at full speed.

Command 0x88: Motor M0 Forward
Compact protocol: **0x88, *motor speed***
Pololu protocol: **0xAA, *device ID*, 0x08, *motor speed***

Command 0x89: Motor M0 Forward (speed + 128; used in 8-bit mode)
Compact protocol: **0x89, *motor speed***
Pololu protocol: **0xAA, *device ID*, 0x09, *motor speed***

Command 0x8A: Motor M0 Reverse
Compact protocol: **0x8A, *motor speed***
Pololu protocol: **0xAA, *device ID*, 0x0A, *motor speed***

Command 0x8B: Motor M0 Reverse (speed + 128; used in 8-bit mode)
Compact protocol: **0x8B, *motor speed***
Pololu protocol: **0xAA, *device ID*, 0x0B, *motor speed***

Motor M1 Commands

Commands 0x8C – 0x8F apply to motor M1. In 8-bit mode, commands 0x8D and 0x8F will set the motor speed to 128+*motor speed*; in 7-bit mode, command 0x8D is identical to command 0x8C and command 0x8F is identical to command 0x8E.

For example, in 8-bit mode, the command packet **0x8C, 0x7F** will set motor M1 speed to 127 out of a maximum of 255, which will result in the motor's turning at half speed. The command packet **0x8D, 0x7F** will set motor M1 speed to 127+128 = 255, which will result in the motor's turning at full speed. In 7-bit mode, both commands will set motor M1 speed to 127 out of a maximum of 127, which will result in the motor's turning at full speed.

Command 0x8C: Motor M1 Forward
Compact protocol: **0x8C,** ***motor speed***
Pololu protocol: **0xAA,** ***device ID*****, 0x0C,** ***motor speed***

Command 0x8D: Motor M1 Forward (speed + 128; used in 8-bit mode)
Compact protocol: **0x8D,** ***motor speed***
Pololu protocol: **0xAA,** ***device ID*****, 0x0D,** ***motor speed***

Command 0x8E: Motor M1 Reverse
Compact protocol: **0x8E,** ***motor speed***
Pololu protocol: **0xAA,** ***device ID*****, 0x0E,** ***motor speed***

Command 0x8F: Motor M1 Reverse (speed + 128; used in 8-bit mode)
Compact protocol: **0x8F,** ***motor speed***
Pololu protocol: **0xAA,** ***device ID*****, 0x0F,** ***motor speed***

6. Cyclic Redundancy Check (CRC) Error Detection

For certain applications, verifying the integrity of the data you're sending and receiving can be very important. Because of this, the qik has optional 7-bit cyclic redundancy checking, which is similar to a checksum but more robust as it can detect some possible errors, such as an extra zero byte, that would not affect a checksum.

When jumper B is in place, cyclic redundancy checking is enabled. In CRC mode, the qik expects an extra byte to be added onto the end of every command packet. The lower seven bits of this byte must be the 7-bit CRC for that packet, or else the qik will set its CRC Error bit in the error byte and ignore the command. The qik will also transmit an additional byte every time it returns data; the lower seven bits of this byte will be the 7-bit CRC for the packet of data the qik is sending you.

A detailed account of how cyclic redundancy checking works is beyond the scope of this document, but you can find a wealth of information using **Wikipedia**. The quick version is that a CRC computation is basically a carryless long division of a CRC "polynomial" 0x91 into your message (expressed as a continuous stream of bits), where all you care about is the remainder. The qik uses CRC-7, which means it uses an 8-bit polynomial (whose most-significant bit, or MSB, must always be 1) and, as a result, produces a 7-bit remainder. This remainder is the lower 7 bits of the CRC byte you tack onto the end of your command packets.

> The CRC implemented on the qik differs from that on the TReX. Instead of being done MSB first, it is LSB first, to match the order in which the bits are transmitted over the serial line. In standard binary notation, the number 0x91 is written as 10010001. However, the bits are transmitted in this order: 1, 0, 0, 0, 1, 0, 0, 1, so we will write it as 10001001 to carry out the computation below.

The CRC-7 algorithm is as follows:

1. Express your 8-bit CRC-7 polynomial and message in binary, LSB first. The polynomial **0x91** is written as **10001001**.
2. Add 7 zeros to the end of your message.
3. Write your CRC-7 polynomial underneath the message so that the LSB of your polynomial is directly below the LSB of your message.
4. If the LSB of your CRC-7 is aligned under a 1, XOR the CRC-7 with the message to get a new message; if the LSB of your CRC-7 is aligned under a 0, do nothing.
5. Shift your CRC-7 right one bit. If all 8 bits of your CRC-7 polynomial still line up underneath message bits, go back to step 4.
6. What's left of your message is now your CRC-7 result (transmit these seven bits as your CRC byte when talking to the qik with CRC enabled).

If you've never encountered CRCs before, this probably sounds a lot more complicated than it really is. Allow me to demonstrate a sample CRC-7 calculation so you can see how this actually works. As an example, we will use a two-byte sequence: 0x83, 0x01 (the command packet to get the PWM configuration parameter byte).

```
Steps 1 & 2 (write as binary, add 7 zeros to the end of the message):
```

```
    CRC-7 Polynomial = [1 0 0 0 1 0 0 1]
    message = [1 1 0 0 0 0 0 1] [1 0 0 0 0 0 0 0] 0 0 0 0 0 0 0 0

Steps 3, 4, & 5:

                  _______________________________________________
    1 0 0 0 1 0 0 1 ) 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
                  XOR 1 0 0 0 1 0 0 1 | | | | | | | | | | | | | |
                      --------------- | | | | | | | | | | | | | |
                        1 0 0 1 0 0 0 1 | | | | | | | | | | | | | |
           shift  ----> 1 0 0 0 1 0 0 1 | | | | | | | | | | | | | |
                        _______________ | | | | | | | | | | | | | |
                              1 1 0 0 0 0 0 0 | | | | | | | | | | | |
                              1 0 0 0 1 0 0 1 | | | | | | | | | | | |
                              _______________ | | | | | | | | | | | |
                                1 0 0 1 0 0 1 0 | | | | | | | | | | |
                                1 0 0 0 1 0 0 1 | | | | | | | | | | |
                                _______________ | | | | | | | | | | |
                                      1 1 0 1 1 0 0 0 | | | | | | | | |
                                      1 0 0 0 1 0 0 1 | | | | | | | | |
                                      _______________ | | | | | | | | |
                                        1 0 1 0 0 0 1 0 | | | | | | | |
                                        1 0 0 0 1 0 0 1 | | | | | | | |
                                        _______________ | | | | | | | |
                                            1 0 1 0 1 1 0 0 | | | | | |
                                            1 0 0 0 1 0 0 1 | | | | | |
                                            _______________ | | | | | |
                                                1 0 0 1 0 1 0 0 | | | | |
                                                1 0 0 0 1 0 0 1 | | | | |
                                                _______________ | | | | |
                                                      1 1 1 0 1 0 0 = 0x17
```

So the full command packet we would send to retrieve the raw channel inputs for all five channels with CRC enabled is: **0x83, 0x01, 0x17**

There are some tricks you can use in your programs to make the CRC calculation much faster. You can find an example of this **Section 6.a**.

6.a. CRC computation in C

The following example program shows how to compute a CRC in the C language. The idea is that the CRC of every possible byte is stored in a lookup table, so that it can be quickly loaded when computing the CRC of a longer message. The individual CRC bytes are XORed together with the C operator ^ to get the final CRC of the message. In the example main() routine, this is applied to generate the CRC byte in the message 0x83, 0x01, that was used in **Section 6**.

```
view plain  print  ?
01. #include <stdio.h>
02.
03. const unsigned char CRC7_POLY                = 0x91;
04. unsigned char CRCTable[256];
05.
06. unsigned char GetCRC(unsigned char val)
07. {
08.     unsigned char j;
09.
10.     for (j = 0; j < 8; j++)
11.     {
12.         if (val & 1)
13.             val ^= CRC7_POLY;
14.         val >>= 1;
15.     }
16.
17.     return val;
18. }
19.
20. void GenerateCRCTable()
21. {
22.     int i, j;
23.
24.     // generate a table value for all 256 possible byte values
25.     for (i = 0; i < 256; i++)
26.     {
27.         CRCTable[i] = GetCRC(i);
28.     }
29. }
30.
31. unsigned char CRC(unsigned char message[], unsigned char length)
32. {
33.     unsigned char i, crc = 0;
34.
35.     for (i = 0; i < length; i++)
36.         crc = CRCTable[crc ^ message[i]];
37.     return crc;
38. }
39.
40. int main()
41. {
42.     unsigned char message[3] = {0x83, 0x01, 0x00};
43.     int i,j;
44.
45.     GenerateCRCTable();
46.     message[2] = CRC(message,2);
47.
48.     for(i=0;i<sizeof(message);i++)
49.     {
50.         for(j=0;j<8;j++)
51.             printf("%d",(message[i]>>j)%2);
52.         printf(" ");
53.     }
54.     printf("\n");
55.     return 0;
56. }
```

7. Troubleshooting

The following are some suggestions for ways you can troubleshoot your qik:

- **Test for life:** With nothing more than logic voltage connected, look for a green LED heartbeat. If you do not see the green LED flashing, you either lack sufficient logic power (e.g. the voltage is out of range or the power supply is too noisy) or your qik is damaged. See **Section 3.a** for more information about logic power requirements.
- **Test your serial connection:** Put on the fixed baud jumper, reset the board, and send 0xBF (191) at 38.4 kbps. You should see the red error

LED turn on. If you then send the command 0x82 (130) at 38.4 kbps, you should see the green LED pulse on (this might be hard to distinguish from the heartbeat LED), the red LED turn off, and you should receive the byte 0x40 (64). If you have a USB-to-TTL-serial adapter, you should try this first with our **Serial Transmitter Utility**, which will help you determine if the problem is with your code.

- **Test your motors:** Try running the qik in **demo mode** first without motors connected and then with motors connected (see **Section 3.d** for more information about demo mode). If the qik experiences problems only with motors connected, this is a sign that your problems are likely due to insufficient motor power or motor-induced noise. Ensure that your motor power supply can provide the current your motors are drawing, and ensure that your motors are not trying to draw more current than the qik can supply (1 A continuous per motor channel, 3 A peak). Make sure your motor power is sufficiently decoupled from your logic power and take steps to limit motor noise (e.g. solder 0.1 uF capacitors across your motor terminals and use short, twisted motor leads).

If your problems persist, please post on our **support forum** or **contact us** directly.