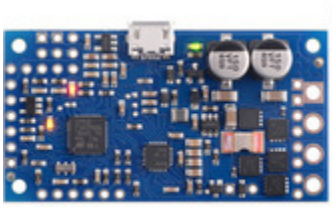
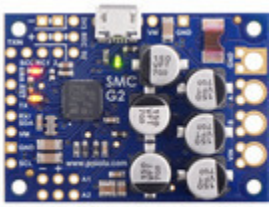
Pololu Simple Motor Controller G2 User's Guide


SMC G2 18v15 or 24v12

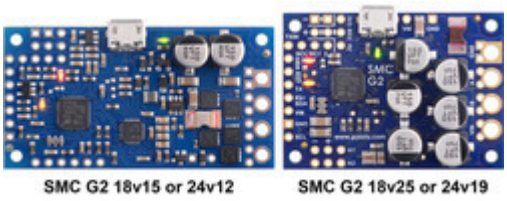


SMC G2 18v25 or 24v19

1. Overview 4
1.1. 18v15 and 24v12 included hardware 8
1.2. 18v25 and 24v19 included hardware 10
1.3. Supported operating systems 12
1.4. Comparison to the original Simple Motor Controllers 12
2. Contacting Pololu 14
3. Getting started 15
3.1. Installing Windows drivers and software 15
3.2. Understanding the control center Status tab 18
3.3. Errors 22
3.5. LED feedback 26
4. Connecting your motor controller 29
4.1. Connecting power and a motor 30
4.2. Serial/I²C interface pins 36
4.3. Connecting a serial device 39
4.4. Connecting an I²C device 41
4.5. Connecting an RC receiver 42
4.6. Connecting a potentiometer or analog joystick 46
5. Configuring your motor controller 51
5.1. Input settings 51
5.1.1. Configuring a limit or kill switch 57
5.2. Motor settings 57
5.3. Advanced settings 62
5.4. Upgrading firmware 65
6. Using the serial and I²C interfaces 66
6.1. Serial and I²C settings 69
6.2. Binary commands 71
6.2.1. Binary command reference 75
6.3. ASCII commands 87
6.3.1. ASCII command reference 90
6.4. Controller variables 96
6.5. Cyclic redundancy check (CRC) error detection 103
6.6. Serial daisy chaining 105
7. Writing PC software to control the Simple Motor Controller G2 108
8. Example code 109

8.1. Example code to run smcg2cmd in C 109
8.2. Example code to run smcg2cmd in Python 110
8.3. Example native USB code in C#, Visual C++, and VB .NET 111
8.4. Example serial code for Arduino 111
8.5. Example serial code for Orangutan 117
8.6. Example serial code for Linux and macOS in C 123
8.7. Example serial code for Windows in C 128
8.8. Example serial code in Python 132
8.9. Example serial code for Linux or macOS in Bash 134
8.10. Example I²C code for Arduino 135
8.11. Example I²C code for Linux in C 137
8.12. Example I²C code for Linux in Python 140
8.13. Example CRC computation in C 142

1. Overview



Side-by-side comparison of the different G2 Simple Motor Controllers.

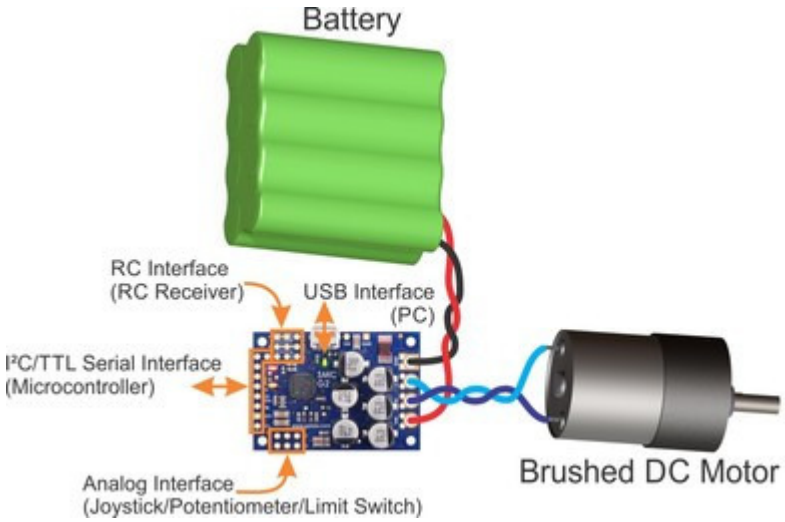
The second-generation G2 Simple Motor Controllers are versatile, general-purpose motor controllers for brushed, DC motors. Wide operating voltage ranges and the ability to deliver up to several hundred Watts in a small form factor make these controllers suitable for many motor control applications. With a variety of supported interfaces—USB for direct connection to a computer, TTL serial and I²C for use with embedded systems, RC hobby servo pulses for use as an RC-controlled electronic speed control (ESC), and analog voltages for use with a potentiometer or analog joystick—and a wide array of configurable settings, these motor controllers make it easy to add basic control of brushed DC motors to a variety of projects. A free configuration utility for Windows simplifies initial setup of the device and allows for in-system testing and monitoring of the controller via USB.

The table below lists the members of the Simple Motor Controller G2 family and shows the key differences among them:

	18v15	**24v12**	**18v25**	**24v19**
Minimum operating voltage:	6.5 V	6.5 V	6.5 V	6.5 V
Recommended max operating voltage:	24 V[1]	34 V[2]	24 V[1]	34 V[2]
Max nominal battery voltage:	18 V	28 V	18 V	28 V
Max continuous current (no additional cooling):	15 A	12 A	25 A	19 A
Dimensions:	2.1″ × 1.1″		1.7″ × 1.2″	
Available with connectors installed?	**Yes**	**Yes**	No	No

1 30 V absolute max.
2 40 V absolute max.

Key features



High-Power Simple Motor Controller G2 18v25 or 24v19 simplified connection diagram.

- Simple bidirectional control of one DC brush motor.
- Five communication or control options:
  1. USB interface for direct connection to a PC.
  2. Logic-level (TTL) serial interface for use with a microcontroller.
  3. I²C interface for use with a microcontroller.
  4. Hobby radio control (RC) pulse width interface for direct connection to an RC receiver or **RC servo controller** **[https://www.pololu.com/category/12/rc-servo-controllers]**.
  5. 0 V to 3.3 V analog voltage interface for direct connection to potentiometers and analog joysticks.
- Simple configuration and calibration over USB with a free configuration program for Windows
- Reverse-voltage protection
- Hardware current limiting with a configurable threshold
- Current sensing

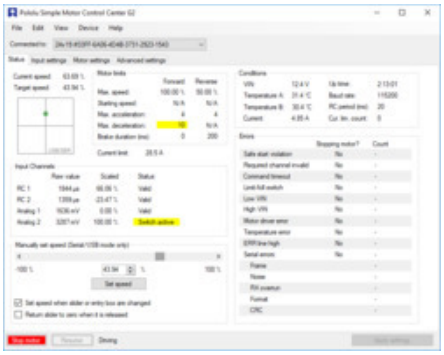
Note: A **USB A to Micro-B cable** **[https://www.pololu.com/product/2073]** (not included) is required to connect this controller to a computer.

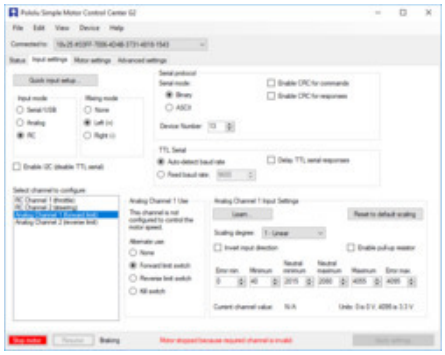
Additional features

- Adjustable maximum acceleration and deceleration to limit electrical and mechanical stress

on the system.

- Adjustable starting speed and maximum speed.
- Option to brake or coast when speed is zero.
- Optional safety controls to avoid unexpectedly powering the motor.
- Input calibration (learning) and adjustable scaling degree for analog and RC signals.
- Under-voltage shutoff with hysteresis for use with batteries vulnerable to over-discharging (e.g. LiPo cells).
- Adjustable over-temperature threshold and response.
- Adjustable PWM frequency from 1.13 kHz to 22.5 kHz (maximum frequency is ultrasonic, eliminating switching-induced audible motor shaft vibration).
- Error LED linked to a digital ERR output, and connecting the error outputs of multiple controllers together optionally causes all connected controllers to shut down when any one of them experiences an error.
- Field-upgradeable firmware.





- **Features of the serial, I²C, and USB interfaces:**
  - Optional CRC error detection to eliminate communication errors caused by noise or software faults.
  - Optional command timeout (shut off motors if communication ceases).
- **Serial features:**
  - Controllable from a computer via serial commands sent to the device's USB virtual serial (COM) port, or via TTL serial through the device's RX/TX pins.
  - TTL serial uses 0 V and 3.3 V on TX, accepts 0 V to 5 V on RX.
  - Supports automatic baud rate detection from 1200 bps to 500 kbps, or can be configured to run at a fixed baud rate.

    - Supports standard compact and Pololu protocols as well as the Scott Edwards Mini SSC protocol and an ASCII protocol for simple serial control from a terminal program.
    - Optional serial response delay for communicating with half-duplex controllers such as the Basic Stamp.
    - Controllers can be easily chained together and to other Pololu serial motor and servo controllers to control hundreds of motors using a single serial line.

- **I²C features:**
    - Compatible with I²C bus voltage levels from 1.8 V to 5 V.

- **USB features:**
    - Full-speed USB interface (12 Mbps)
    - Example code in C#, Visual Basic .NET, and Visual C++ is available in the **Pololu USB Software Development Kit [https://www.pololu.com/docs/0J41]**

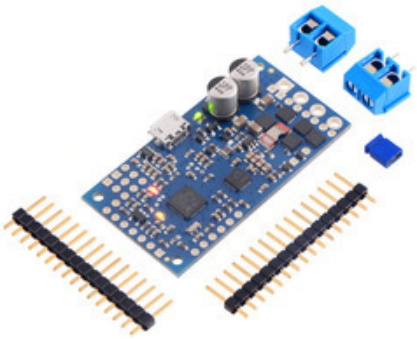
- **RC features:**
    - 1/4 µs pulse measurement resolution.
    - Works with RC pulse frequencies from 10 to 333 Hz.
    - Configurable parameters for determining what constitutes an acceptable RC signal.
    - Two RC channels allow for single-stick (mixed) motor control, making it easy to use two simple motor controllers in tandem on an RC-controlled differential-drive robot.
    - RC channels can be used in any mode as limit or kill switches (e.g. use an RC receiver to trigger a kill switch on your autonomous robot).
    - Battery elimination circuit (BEC) jumper can power the RC receiver with 5 V or 3.3 V.

- **Analog features:**
    - 0.8 mV (12-bit) measurement resolution.
    - Works with 0 to 3.3 V inputs.
    - Optional potentiometer/joystick disconnect detection.
    - Two analog channels allow for single-stick (mixed) motor control, making it easy to use two simple motor controllers in tandem on a joystick-controlled differential-drive robot.
    - Analog channels can be used in any mode as limit or kill switches.

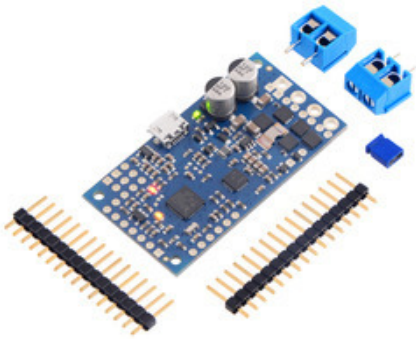
Note: This guide only applies to the G2 Simple Motor Controllers, which have blue circuit boards. If you have one of the first-generation Simple Motor Controllers, which have green circuit boards, you can find their user's guide **here** [https://www.pololu.com/docs/0J44].

Warning: Take proper safety precautions when using high-power electronics. Make sure you know what you are doing when using high voltages or currents! During normal operation, this product can get hot enough to burn you. Take care when handling this product or other components connected to it.

1.1. 18v15 and 24v12 included hardware



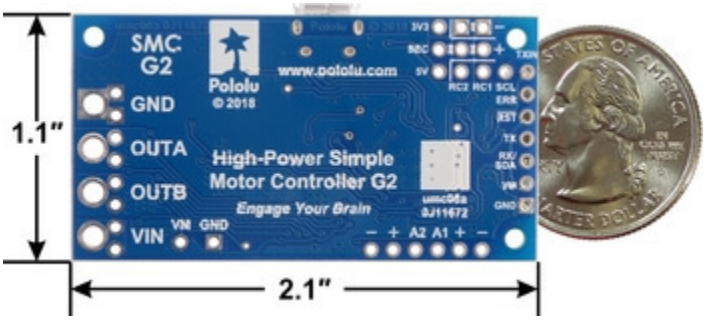
High-Power Simple Motor Controller G2 18v15 with included hardware.



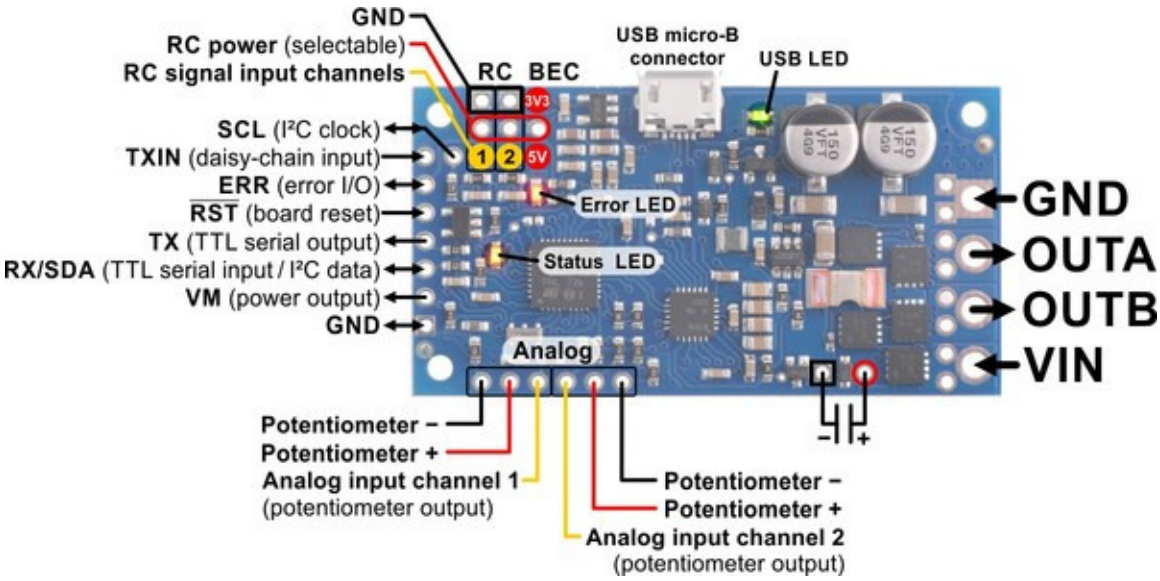
High-Power Simple Motor Controller G2 24v12 with included hardware.

High-Power Simple Motor Controller G2 18v15 with connectors soldered.

High-Power Simple Motor Controller G2 24v12 with connectors soldered.



High-Power Simple Motor Controller G2 18v15 or 24v12, bottom view with dimensions.



Pinout diagram of the High-Power Simple Motor Controller G2 18v15 or 24v12.

The 18v15 and 24v12 versions are available with connectors included but not soldered in or with the connectors pre-installed.

The terminal blocks are only rated for 16 A, so for higher-power applications, we recommend soldering thick wires directly to the board.

These files provide further documentation of the hardware design of the Simple Motor Controller G2 18v15 and 24v12:

- **Dimension diagram [https://www.pololu.com/file/0J1600/pololu-smc-g2-18v15-or-24v12-dimensions.pdf]** (424k pdf)
- **3D model [https://www.pololu.com/file/0J1602/pololu-smc-g2-18v15-or-24v12.step]** (12MB step)

- **Drill guide** [https://www.pololu.com/file/0J1601/umc08a-drill.dxf] (100k dxf)

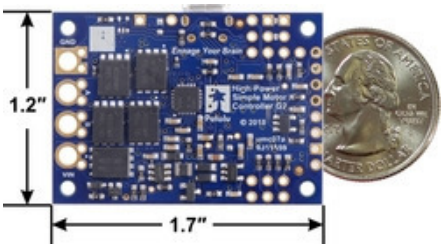
1.2. 18v25 and 24v19 included hardware

High-Power Simple Motor Controller G2 18v25 with included hardware.

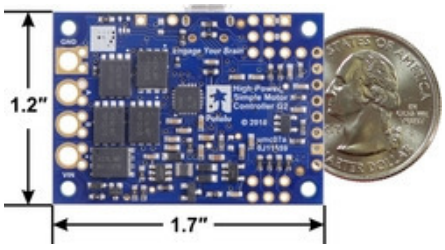
High-Power Simple Motor Controller G2 24v19 with included hardware.

High-Power Simple Motor Controller G2 18v25 with connectors soldered.

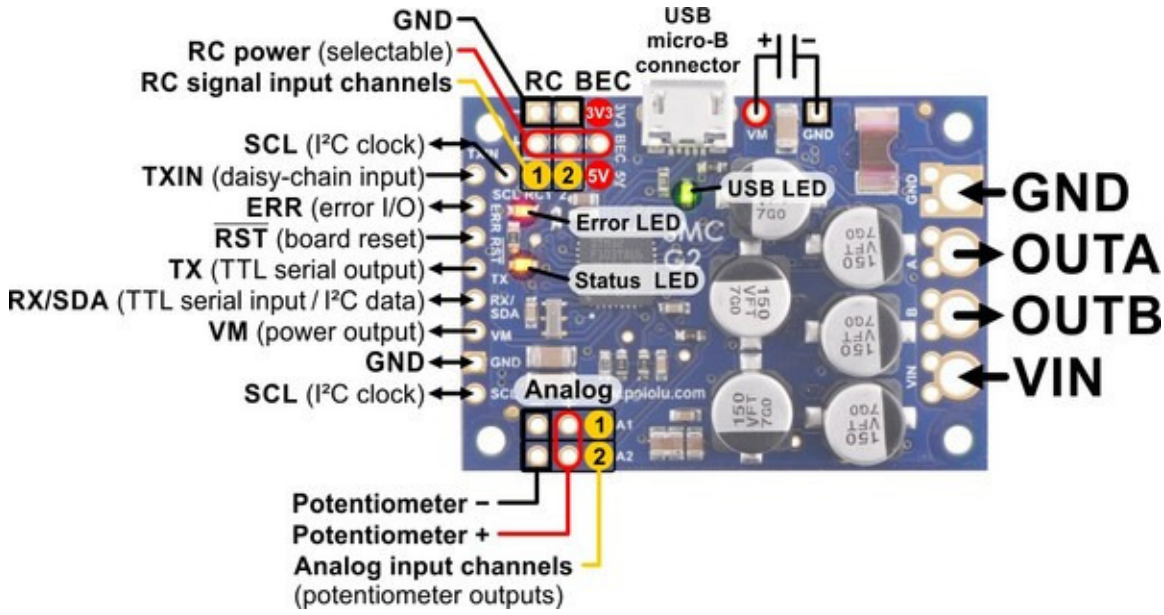
High-Power Simple Motor Controller G2 24v19 with connectors soldered.



High-Power Simple Motor Controller G2 18v25, bottom view with dimensions.



High-Power Simple Motor Controller G2 24v19, bottom view with dimensions.



Pinout diagram of the High-Power Simple Motor Controller G2 18v25 or 24v19.

The 18v25 and 24v19 versions come with connectors included but not soldered.

The terminal blocks are only rated for 16 A, so for higher-power applications, we recommend soldering thick wires directly to the board.

High-Power Simple Motor Controller G2 18v25 or 24v19 with thick wires soldered.

These files provide further documentation of the hardware design of the Simple Motor Controller G2 18v25 and 24v19:

- **Dimension diagram [https://www.pololu.com/file/0J1603/pololu-smc-g2-18v25-or-24v19-dimensions.pdf]** (379k pdf)
- **3D model [https://www.pololu.com/file/0J1605/pololu-smc-g2-18v25-or-24v19.step]** (12MB step)
- **Drill guide [https://www.pololu.com/file/0J1604/umc07a-drill.dxf]** (102k dxf)

1.3. Supported operating systems

The Simple Motor Controller G2 configuration software works on Windows 7, Windows 8, and Windows 10.

We do not currently provide configuration software for Linux or macOS, but **Section 8.6** has some example code for controlling the Simple Motor Controller via its USB serial port from Linux or macOS.

1.4. Comparison to the original Simple Motor Controllers

This section lists most of the things you should consider if you have an existing application using the original Simple Motor Controller controller (green board) and are considering upgrading to a Simple Motor Controller G2 (blue board).

Motor driver changes

The G2 Simple Motor Controllers have configurable hardware current limiting: when the motor current exceeds a configurable threshold, the motor driver uses current chopping to actively limit it. The current limit threshold can be configured ahead of time, and can also be changed dynamically via serial, I²C, or USB. The SMC G2 can also measure the current being drawn by the motor.

The G2 Simple Motor Controllers do not support variable braking. Instead, they can either do full braking or full coasting when the speed is zero.

Physical connection changes

You need to keep some things in mind when updating the physical connections of an existing application:

- The Simple Motor Controller G2 uses a USB Micro-B connector (the original controllers used Mini-B).
- The through-hole capacitors have been replaced with SMT capacitors.
- The SMC G2 18v15 and 24v12 boards have the same size and nearly the same layout as the original SMC 18v15 and 24v12. An SCL pin was added.

- The SMC G2 18v25 and 24v19 are significantly smaller than the original SMC 18v25 and 24v23, and the motor/power connections are different. Two SCL pins were added, but otherwise the control I/O pins are the same.

Configuration and software changes

There are several changes to keep in mind when configuring the Simple Motor Controller G2 or updating any software that communicates with it:

- The Simple Motor Controller G2 uses different configuration software from the original Simple Motor Controllers.
- We do not currently provide a Linux version of the SMC G2 configuration software.
- The Simple Motor Controller G2 serial protocol is generally a superset of the original serial protocol, so in many cases, serial interface software running on a microcontroller or computer (using the controller's RX and TX lines or its virtual USB serial ports) will not need to be modified to work with the Simple Motor Controller G2.
- The Simple Motor Controller G2 native USB interface uses different product IDs and a different arrangement of settings in memory. One command was added (Set current limit). Most of the commands remain unchanged.
- Since the Simple Motor Controller G2 does not have variable braking, any "Motor brake" command received via serial or I²C that specifies a non-zero brake amount will be interpreted as a request for full braking. This also applies to the USB "Set speed" command.

New features

- The SMC G2 boards have reverse voltage protection.
- The SMC G2's new I²C interface provides another option for connecting to a microcontroller.
- The SMC G2 can be configured to send CRC bytes for serial responses without requiring them on serial commands.
- The SMC G2's native USB interface implements Microsoft OS 2.0 Descriptors, so it will work on Windows 8.1 or later without needing any drivers. (The USB serial ports will work on Windows 10 or later without drivers.)

2. Contacting Pololu

We would be delighted to hear from you about any of your projects and about your experience with the G2 Simple Motor Controllers. You can **contact us** **[https://www.pololu.com/contact]** directly or post on our **forum** **[https://forum.pololu.com/]**. 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. Getting started

3.1. Installing Windows drivers and software

To install the drivers for the Simple Motor Controller G2 on a computer running Microsoft Windows, follow these steps:

1. Download the **Simple Motor Controller G2 Software and Drivers for Windows [https://www.pololu.com/file/0J1599/smc-g2-1.0.0-windows.zip]** (550k zip)
2. Open the ZIP archive and run *setup.exe*. The installer will guide you through the steps required to install the Simple Motor Control Center G2, the Simple Motor Controller G2 command-line utility (smcg2cmd), and the Simple Motor Controller drivers on your computer. If the installer fails, you might have to extract all the files to a temporary directory, right click *setup.exe*, and select "Run as administrator".
3. During the installation, Windows will ask you if you want to install the drivers. Click "Install" to proceed.
4. After the installation has completed, plug the Simple Motor Controller G2 into your computer via USB. Windows should recognize the controller and load the drivers that you just installed.
5. Open your Start Menu and search for "G2". Select the "Simple Motor Control Center G2" shortcut (in the Pololu folder) to launch the software.
6. In the upper left corner of the window, where it says "Connected to:", make sure that it shows something like "18v25 #33FF-7006-4D4B-3731-4818-1543". This indicates the version and serial number of the controller that the software has connected to. If it says "Not connected", see the troubleshooting section below.

The SMC G2's native USB interface implements Microsoft OS 2.0 Descriptors, so it will work on Windows 8.1 or later without needing any drivers. The USB serial ports will work on Windows 10 or later without drivers.

The Simple Motor Controller G2 software consists of two programs:

- The Simple Motor Control Center G2 is a graphical user interface (GUI) for configuring the controller, viewing its status, and controlling it manually. You can find the configuration utility in your Start Menu by searching for it or looking in the Pololu folder.
- The Simple Motor Controller G2 Command-line Utility (smcg2cmd) is a command-line utility that can do most of what the GUI can do, and more. You can open a Command Prompt and type `smcg2cmd` with no arguments to see a summary of its options.

USB troubleshooting for Windows

If the Simple Motor Controller G2 software cannot connect to your controller after you plug it into the computer via USB, the tips here can help you troubleshoot the SMC's USB connection.

If you are using the Simple Motor Control Center G2, try opening the "Connected to:" drop-down box to see if there are any entries in the list. If there is an entry, try selecting it to connect to it.

Make sure you have a Simple Motor Controller G2 (blue board). The G2 software does not work with the original Simple Motor Controllers (green boards). If you have one of those products, you should refer to its user's guide instead of this user's guide.

Make sure you are using software that supports the Simple Motor Controller G2. The original Simple Motor Control Center does not work with the SMC G2. The SMC G2 controllers have new USB product IDs. Third-party software for the older controllers might need to be updated, depending on how the software works. If you are a developer of such software, see **Section 1.4**.

If you have connected any electronic devices to your Simple Motor Controller besides the USB cable, you should disconnect them.

You should look at the LEDs of the Simple Motor Controller. If the LEDs are off, then the controller is probably not receiving power from the USB port. If the green LED is flashing very briefly once per second, then the controller is receiving power from USB, but it is not receiving any data. These issues can be caused by using a broken USB port, using a broken USB cable, or by using a USB charging cable that does not have data wires. Using a different USB port and a different USB cable, both of which are known to work with other devices, is a good thing to try. Also, if you are connecting the controller to your computer via a USB hub, try connecting it directly.

If the controller's green LED is on all the time or flashing slowly, but you can't connect to it in the software, then there might be something wrong with your computer. A good thing to try is to unplug the controller from USB, reboot your computer, and then plug it in again.

If that does not help, you should go to your computer's Device Manager and locate all the entries for the Simple Motor Controller. Be sure to look in these categories: "Other devices", "Ports (COM & LPT)", and "Universal Serial Bus devices".

If the driver for the Simple Motor Controller's native USB interface is working, you should see an entry in the "Universal Serial Bus devices" category named something like "Pololu High-Power Simple Motor Controller G2 18v15" (or the corresponding name if you have a different version).

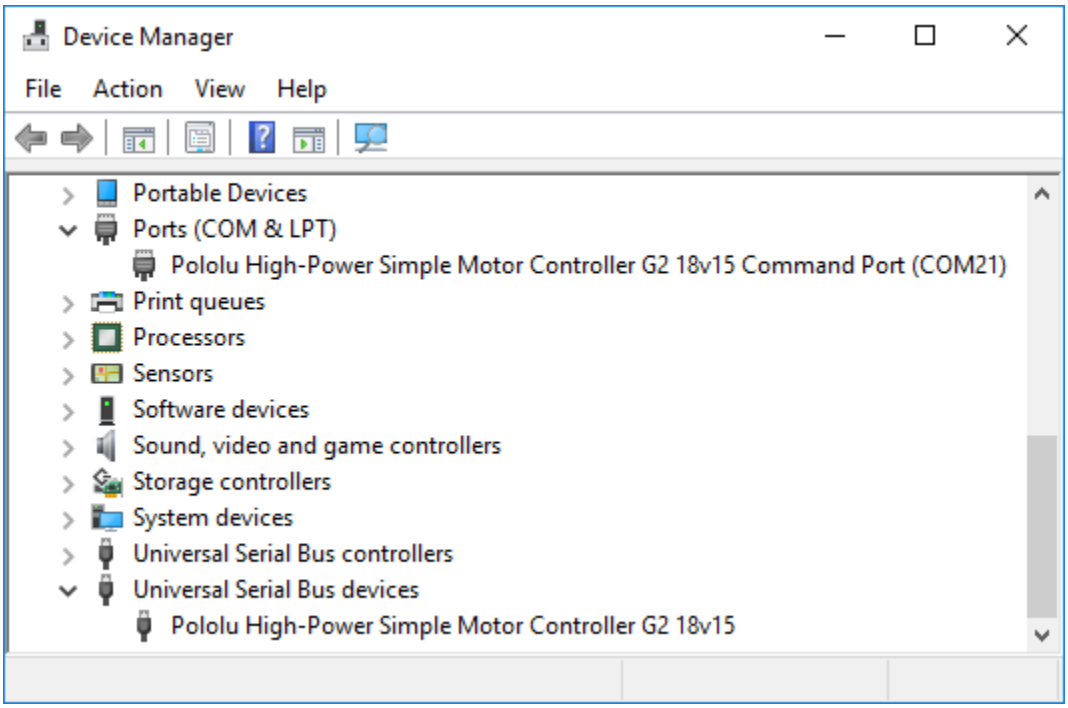
If the drivers for the Simple Motor Controller's USB serial ports are working, you should see an entry in the "Ports (COM & LPT)" category named something like "Pololu High-Power Simple Motor Controller

G2 18v15 Command Port". The serial port might be named "USB Serial Device" instead if you are using Windows 10 or later and you plugged the controller into your computer before installing our drivers for it. The generic name in the Device Manager will not prevent you from using the port, but we recommend fixing the name by right-clicking on each "USB Serial Device" entry, selecting "Update Driver Software…", and then selecting "Search automatically for updated driver software". Windows should find the drivers you already installed, which contain the correct name for the port.

If any of the entries for the Simple Motor Controller in the Device Manager has a yellow triangle displayed over its icon, you should double-click on the entry to get information about the error that is happening.

If you do not see entries for the Simple Motor Controller in the Device Manager, then you should open the "View" menu and select "Devices by connection". Then expand the entries until you find your computer's USB controllers, hubs, and devices. See if there are any entries in the USB area that disappear when you unplug the controller. This might give you important information about what is going wrong.

Do **not** attempt to fix driver issues in your Device Manager using the "Add legacy hardware" option. This is only for older devices that do not support Plug-and-Play, so it will not help. If you already tried this option, we recommend unplugging the Simple Motor Controller from USB and then removing any entries you see for Simple Motor Controller by right-clicking on them and selecting "Uninstall". Do **not** check the checkbox that says "Delete the driver software for this device".

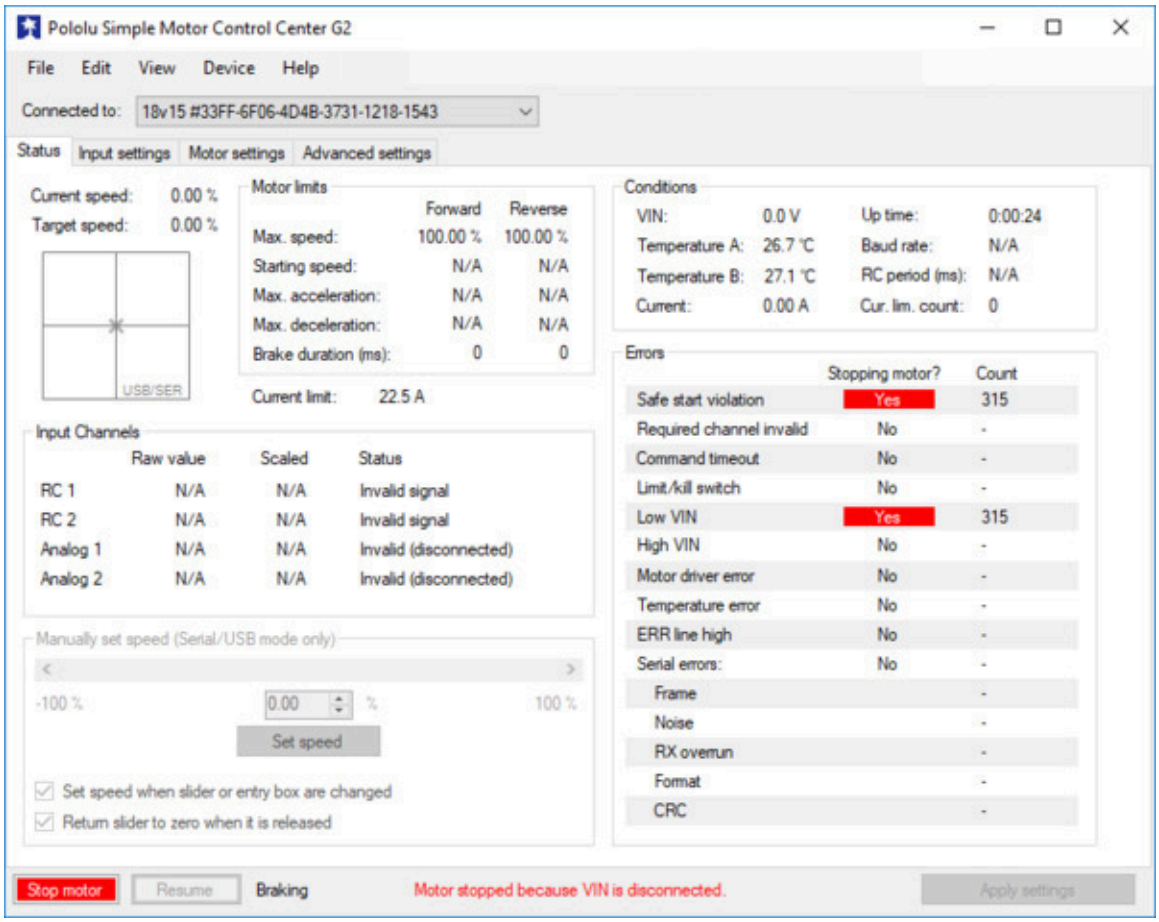


Windows 10 Device Manager showing the Simple Motor Controller G2.

3.2. Understanding the control center Status tab

After installing the software for the Simple Motor Controller G2, it is a good idea to run the Simple Motor Control Center G2 software and look at the Status tab. The Status tab lets you monitor the status of your motor controller in real time and control the speed of the motor. The Status tab also shows what errors and limits are affecting your motor controller so it can help you quickly troubleshoot any issues you are having.

To use the Status tab, you should connect your Simple Motor Controller to your PC using a USB cable (not included) and run the Pololu Simple Motor Control Center. This is what the Status tab should look like initially, before you have modified any settings or connected anything to the Simple Motor Controller (besides USB):



The Status tab in the Simple Motor Control Center G2 should look like this when you first connect the controller to the PC.

Target speed and current speed

The **Target speed** is the speed that the motor controller is trying to achieve. The target speed source is determined by the settings in the "Input settings" tab, and can come from serial/I²C/USB commands, analog voltages, or RC signals.

The **Current speed** is the speed at which the controller is currently your driving your motor. There are several reasons why the current speed might be different from the target speed: errors, acceleration limits, deceleration limits, brake duration, maximum speed limits, starting speed limits, and gradual temperature-based speed limiting. If any of these things are affecting the current speed, the appropriate part of the Status tab will be highlighted to let you know. Anything that is stopping the motor completely will be highlighted in red. Anything that is limiting the speed of the motor will be highlighted in yellow. While hardware current limiting can affect the power delivered to the motor, it does not affect this "current speed" reading.

The Simple Motor Controller represents speeds internally as a number from -3200 (full reverse) to 3200 (full forward). However, by default the speeds in the Status Tab are displayed as percentages so -3200 (full reverse) is shown as -100.00% and 3200 (full forward) is shown as 100.00%.

Below the Target Speed label is a two-dimensional diagram that represents the values of the inputs that are used to set the Target Speed. This diagram is especially useful in RC or Analog mode with Mixing enabled because it graphically shows you the value of both input channels and makes it easier to tell how well the Simple Motor Controller is calibrated for your controller.

Motor limits

The "Motor limits" box in the Status tab shows the current values of the limits on the movement of the motor. These limits will be equal to the hard motor limits specified in the "Motor settings" tab, unless you have temporarily changed the motor limits using the command-line utility (smcg2cmd) or a serial/I²C/USB command. For more information on these limits, see **Section 5.2**.

Current limit

The "Current limit" is displayed below the "Motor limits" box. This is the current value of the hardware current limit.

Input channels

The "Input channels" box in the Status tab shows the current status of the RC and Analog input channels of the device.

The **Raw value** is the raw, unscaled value of the input channel. For RC channels, the raw value is the width of pulses received on the input line (RC1 or RC2). It is typically between 1000 µs and 2000 µs, and it is stored internally as an integer in units of quarter-microseconds (6000 corresponds to 1500 µs). For analog channels, the raw value is the average voltage measured on the input line (A1 or A2). It is always between 0 mV and 3300 mV, and it is stored internally as a 12-bit integer (0 corresponds to 0 mV while 4095 corresponds to 3300 mV).

The **Scaled value** is a number between −3200 and 3200 that is determined entirely by the **raw value** and the scaling parameters in the "Input settings" tab. If the scaling parameters are set up correctly, then the scaled value should be 0 when the input is in its neutral position (if it has a neutral position), and they should be ±100 % (±3200 internally) when the input is moved to either extreme.

The **Status** column summarizes the state of each channel. Here are the different things you might see in this column:

- **Valid:** There is an RC or analog input connected to this channel and it is working.
- **Invalid (disconnected):** This message is shown for analog channels when the controller

detects that they are disconnected. If you do not intend to use this channel, you do not need to worry about this message. Otherwise, to correct this situation, make sure that all three pins of your potentiometer or analog joystick are connected correctly to the three analog interface pins (see **Section 4.6**). The controller toggles the power supply on the Analog + pins in order to detect when your potentiometer is disconnected. This feature can be turned off in the Advanced tab, in which case you will not see the "Invalid (disconnected)" message.

- **Invalid signal:** This message is shown for RC channels when the controller detects no signal or a bad signal on the RC input. If you do not intend to use this channel, you do not need to worry about this message. Otherwise, to correct this situation, make sure that your RC receiver is powered and connected correctly (see **Section 4.5**), and check your RC pulse detection settings in the "Advanced settings" tab.
- **Invalid (too high)** and **Invalid (too low):** These messages are shown for analog channels when the voltage read on the A1 or A2 pin is outside of the normal range, as specified by the "Error min" and "Error max" parameters for that channel in the "Input settings" tab. To correct this error, you can re-configure the range of your analog input by clicking the "Learn…" button for that channel, or you can manually adjust the scaling parameters.
- **Invalid (high signal)** and **Invalid (low signal):** These messages are shown for RC channels when the pulse width measured on the RC1 or RC2 pin is outside of the normal range as specified by the "Error min" and "Error max" parameters for that channel in the "Input settings" tab. To correct this error, you can re-configure the range of your RC input by clicking the "Learn…" button for that channel, or you can manually adjust the scaling parameters.

Conditions

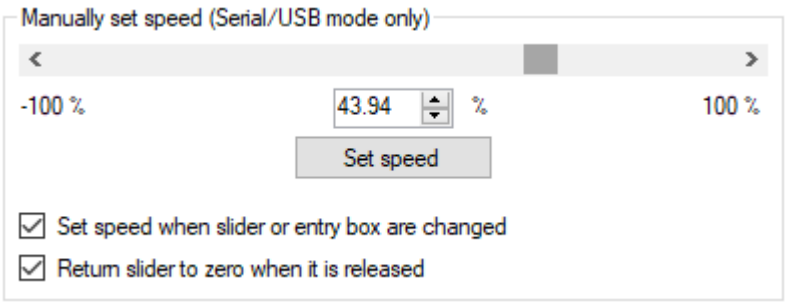
The Conditions box in the Status tab shows miscellaneous information about the current state of the controller:

- **VIN:** This is the voltage of your power supply, measured on the VIN line. When your power supply is disconnected, this should read 0.0 V. This reading is continually compared to the VIN thresholds in the Advanced Settings tab and will generate an error and shut down the motor if it passes these thresholds. This allows a properly configured controller to avoid over-discharging your batteries.
- **Temperature A** and **Temperature B:** These are measurements of the temperature of two different points on the circuit board. These readings are used prevent damage to the device by shutting down the motor when the board gets too hot (the over-temperature threshold is can be adjusted in the "Advanced settings" tab). Please note that this product can get hot enough to burn you during normal operation. Take care when handling this product or other components connected to it. Parts of the board can be significantly hotter than this reading, so you should **not** rely on this temperature reading when deciding whether it is safe to touch

the board.

- **Up time:** This is the total amount of time that the controller has been running since its last reset or power-up. The up time reading can be used to help identify if the controller has reset unexpectedly. You can determine the cause of a reset by looking at the pattern of the yellow LED (see **Section 3.5**), or you can look in the Device Information window, available from the Device menu. The "Up time" reading will overflow back to zero after 49.7 days.
- **Baud rate:** This is the current baud rate that the device is using on the TTL serial interface (RX and TX lines) in units of bits per second (bps). By default, the device is configured to auto-detect the baud rate, so this value will be "N/A" until the baud rate is detected. After a 0xAA byte is received on the RX line, the device will detect the baud rate and you can see it here. *Please note that the baud rate display in the Status tab has nothing to do with the USB virtual COM port; it doesn't matter what baud rate you use when connecting to the virtual COM port*.
- **RC period:** This is the period of the RC signal on the RC1 input channel. You can use this reading to help you make the RC period settings in the "Advanced settings" tab more strict so that the controller can better identify bad RC signals. If the signal on RC1 is invalid, this reading is reported as "N/A".
- **Current:** This is a measurement of the current going through the motor.
- **Cur. lim. count:** This is short for "Current limit count", and it reports how many times the controller has detected that the hardware current limiting activated in order to limit the current through the motor. You can clear this count by opening the "Device" menu and selecting "Clear counts".

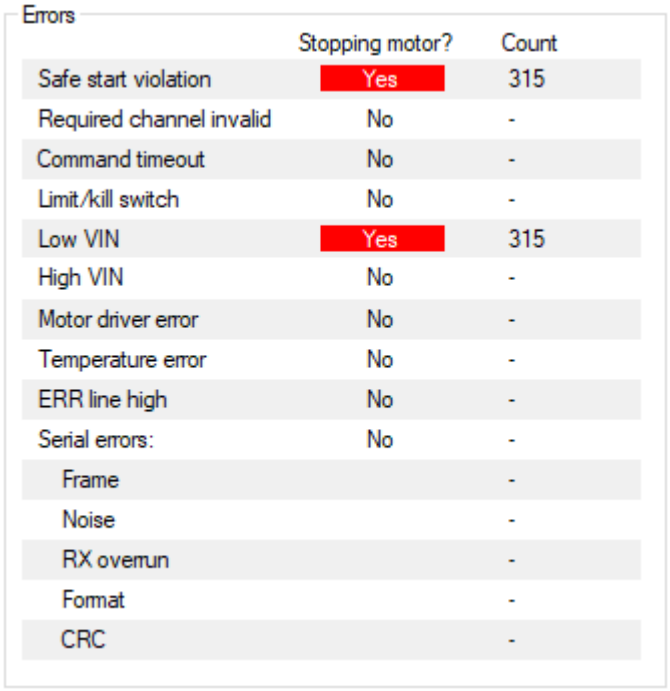
Manually set speed (Serial/USB mode only)



The "Manually set speed" box in Status tab allows you to control the speed of your motor over USB by using a scrollbar or by typing in a speed. To use this feature, the input mode (configured in the "Input settings" tab) must be USB/Serial, and there must be no errors currently stopping the motor. You will need to press the Resume button if you have not disabled safe start or if you previously pressed the "Stop motor" button.

3.3. Errors

The Simple Motor Controller G2 has several features that stop the motor when something is going wrong. These are called *errors*, and they can help protect your project from damaging itself. Whenever you are having an issue with the controller, you should first check to see what (if any) errors are occurring. You can get information about the errors by:



The Errors box in the Status tab of the Simple Motor Control Center G2 shows problems that are stopping your motor.

- Checking the Errors box in the Status tab of the Simple Motor Control Center. This is recommended because it gives you the most information, including a running count of how many times the error has been reported.
- Running the command-line utility (just type `smcg2cmd -s` at the command line).
- Looking at the red LED on the device. It will be lit if there are any errors stopping your motor.
- Writing PC software or using a microcontroller to send the "Get errors" serial/I²C command.
- Using a microcontroller to measure the voltage on ERR pin. This pin is linked to the red LED so it should go high (3.3 V) when there is an error stopping your motor and low (0 V) otherwise.

All the errors are explained below:

- **Safe start violation:** Safe start is a feature that helps prevent the motor from starting up unexpectedly. This feature is enabled by default, but can be disabled in the "Advanced settings" tab. The behavior of safe start depends on what input mode you are using.

  In Serial/USB input mode, the safe start violation error occurs whenever any other error is

stopping the motor. After all the other errors have been fixed, you can clear the safe start violation error by pressing the Resume button (which issues a native USB command) or using a serial command.

In Analog or RC input mode, the safe start violation error occurs whenever the motor is stopped because of an error AND the inputs that control the speed of the motor are not near their neutral positions. This helps prevent the situation where there might be an error stopping your motor (such as a disconnected battery), and the motor starts running at a high speed when you fix the error. To clear the safe start violation error, move all the inputs that control the speed of the motor to their neutral positions (the sum of the absolute values of their scaled values must be less than 8%).

- **Required channel invalid:** This error occurs whenever any required RC or analog channel is invalid. This error helps ensure that your motor will stop if you accidentally disconnect your joystick, potentiometer or RC receiver. A channel is *invalid* if it is disconnected, or has a value that is out of range. A channel is *required* if it controls the speed of the motor or it is configured as a limit switch or kill switch. By default, there are no required channels because the input mode is serial and no limit or kill switches have been configured. You can check the "Input settings" tab to see which channels are required. Channels that are required and invalid are highlighted in red in the "Input channels" box of the Status tab so you can quickly see which channel is causing this error.
- **Command timeout:** This error occurs if you are controlling your motor using a microcontroller or a PC (input mode is Serial/USB) and the (configurable) time period has elapsed with no valid serial, I²C, or USB commands being received by the controller. The purpose of this error is to ensure that your motor will stop if the software talking to the controller crashes or if the communications link is broken. All valid serial and I²C commands clear this error. The native USB commands for setting the speed and exiting safe start also clear this error. By default, this error is disabled, but it can be enabled from the "Advanced settings" tab by setting a non-zero command timeout value.
- **Limit/kill switch:** This error occurs when a limit or kill switch channel stops the motor. More specifically, it occurs in three cases: when a kill switch is active, when a forward limit switch is active AND the target speed is positive, or when a reverse limit switch is active AND the target speed is negative. A limit/kill switch is considered active if its scaled value is above 50 %. If you are using a limit switch and your input mode is Serial/USB, you will need to check the Count column in the Status tab to see this error because in Serial/USB mode the Target Speed gets set to 0 whenever there is an error.
- **Low VIN:** This error occurs whenever your power supply's voltage is too low or it is disconnected. If you set the correct thresholds in the "Advanced settings" tab, this error will prevent you from over-discharging your battery.

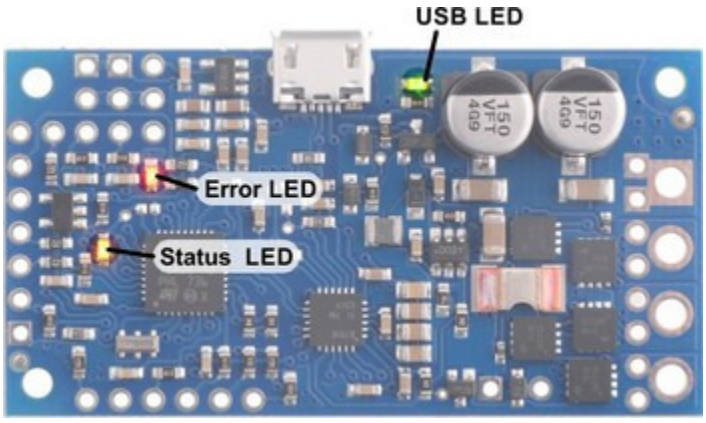
- **High VIN:** This error occurs whenever your power supply's voltage is too high. You can set the threshold voltage in the "Advanced settings" tab.
- **Motor driver error:** This error occurs whenever the motor driver chip reports an under-voltage, over-current, or over-temperature error.
- **Over temperature:** This error occurs whenever a reading from one of the temperature sensors is too high. You can see the temperature readings in the "Conditions" box of the Status tab. The behavior of this error and the threshold temperatures can be configured in the "Advanced settings" tab.
- **ERR line high:** This error occurs whenever there are no other errors but the voltage on the ERR line is high (2.3 V to 5 V). This error allows you to connect the error lines of two Simple Motor Controllers together and have both of them stop when either one experiences an error. This error can be disabled in the Advanced Settings tab.
- **Serial errors:** Serial errors are recorded whenever something goes wrong with the serial or I²C communication, either on the TX/RX/SDA/SCL lines or on the USB virtual COM port. If the input mode is Serial/USB, then a serial error will stop the motor from running until a valid serial command is received, or the Resume button is pressed, or the native USB "Set speed" or "Exit safe start" commands are sent. If you are using serial and have not disabled safe start mode, you will need to send the "Exit safe-start" command followed by a "Set speed" command to recover from an error and get the motor running again. If you are using serial and *have* disabled safe start, the motor will start driving as soon as a valid "Set speed" command is received. These are the types of serial errors that are recorded:
  - **Frame:** This is error occurs when a de-synchronization or excessive noise on the RX line is detected.
  - **Noise:** This error occurs when noise is detected on the RX line.
  - **RX overrun:** This error occurs when the buffer for storing bytes received on the RX line is full and data was lost as a result. This should not occur during normal operation.
  - **Format:** This error occurs if the serial or I²C bytes received do not obey the protocol specified in this guide. If you get this error, check the bytes you are sending carefully, and compare them to the examples provided.
  - **CRC:** This error occurs if you have enabled cyclic redundancy check (CRC) for serial commands, but the CRC byte received was invalid. CRC helps prevent the motor controller from accidentally performing unwanted actions when it is receiving commands over a noisy serial link. If you get this error, check your algorithm for calculating CRCs and check the quality of your serial signal at the RX pin.

The count that is displayed next to each error in the Simple Motor Control Center G2 indicates how

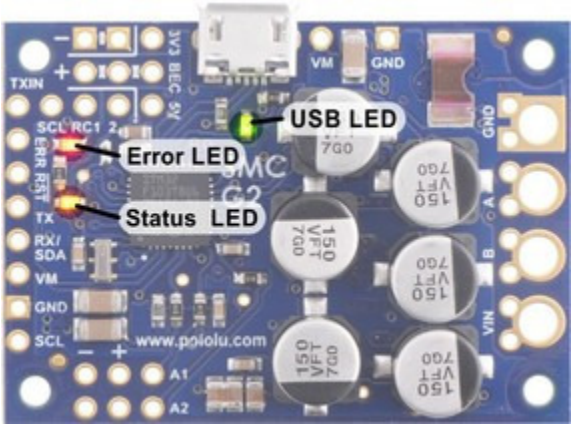
many times the occurrence of the error has been reported. You can clear these counts by opening the "Device" menu and selecting "Clear counts".

3.5. LED feedback

The Simple Motor Controller G2 has three indicator LEDs that provide feedback about the current state of the controller. The LEDs can tell you whether an error is occurring, whether the USB connection is active, what direction the motor is driving, and much more.



High-Power Simple Motor Controller G2 18v15 or 24v12 LEDs.



High-Power Simple Motor Controller G2 18v25 or 24v19 LEDs.

Green USB LED

This LED indicates the USB status of the device. When the controller is not connected to a computer via the USB cable, the green LED will always be off. When you connect the controller to USB, the green LED starts blinking slowly. The blinking continues until the controller receives a particular

message from the computer indicating that the controller's USB drivers are installed correctly (see **Section 3.1** for driver installation instructions). After the controller gets this message, the green LED turns solidly on, except for brief flickers whenever there is USB activity. The Simple Motor Control Center G2 software constantly streams data from the controller, so when the control center is running and connected to the Simple Motor Controller, the green LED will flicker constantly.

Red Error LED

This LED turns on whenever there is an error stopping the motor (see **Section 3.3**). The red LED is tied directly to the active-high output **ERR**, which allows the error status to be monitored by an external device such as a microcontroller. When no errors are stopping the motor, the error LED is off and the ERR pin is pulled low. See **Section 4.3** for more information about the ERR pin and the error LED.

Yellow Status LED

This LED helps you visually identify the state of the device, which can be useful when the controller is not connected to the Control Center. On start-up, the status LED briefly flashes a pattern indicating the source of the last reset (see the Reset Flags variable in **Section 6.4** for more information):

- 8 blinks over the first two seconds after start-up indicates that the external **$\overline{\text{RST}}$** pin was driven low to reset the controller.
- 3 blinks over the first two seconds after start-up indicates that the controller last reset because logic power got too low (power was disconnected or the controller browned out).
- Rapid flickering for the first two seconds after start-up indicates that the controller was reset by a software fault or by a firmware upgrade.

This startup behavior can help you detect if your Simple Motor Controller is browning out and resetting unexpectedly (as can happen if your input voltage drops due to high power demands or electrical noise).

After the start-up phase ends, the status LED primarily gives feedback about the motor driver outputs:

- An even blinking pattern of on for 2/3 s and off for 2/3 s indicates that the controller is not driving the motor and has not yet detected the baud rate. This pattern only occurs when the controller is in USB/serial mode with automatic baud detection enabled and helps you determine when you have established communication between a TTL serial source and the Simple Motor Controller.
- A brief flash once per second indicates that the controller is not driving the motor. If the controller is in Serial/USB mode with automatic baud detection enabled, this pattern additionally indicates that the Simple Motor Controller has successfully learned the TTL serial baud rate.

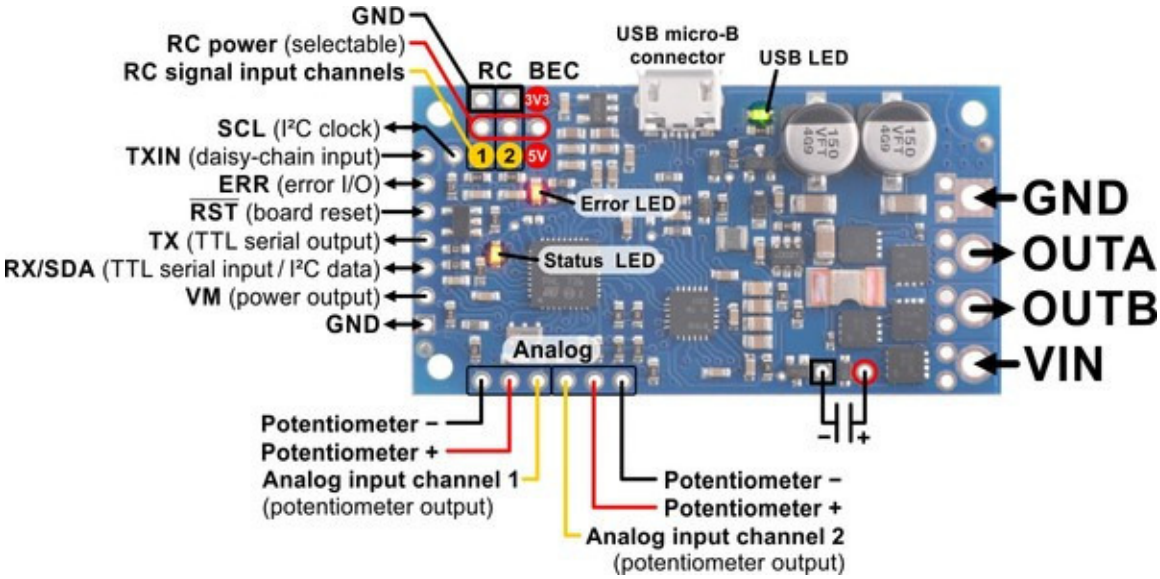
- A repeating, gradual increase in brightness every second indicates that the controller is driving the motor forward.
- A repeating, gradual decrease in brightness every second indicates that the controller is driving the motor in reverse.

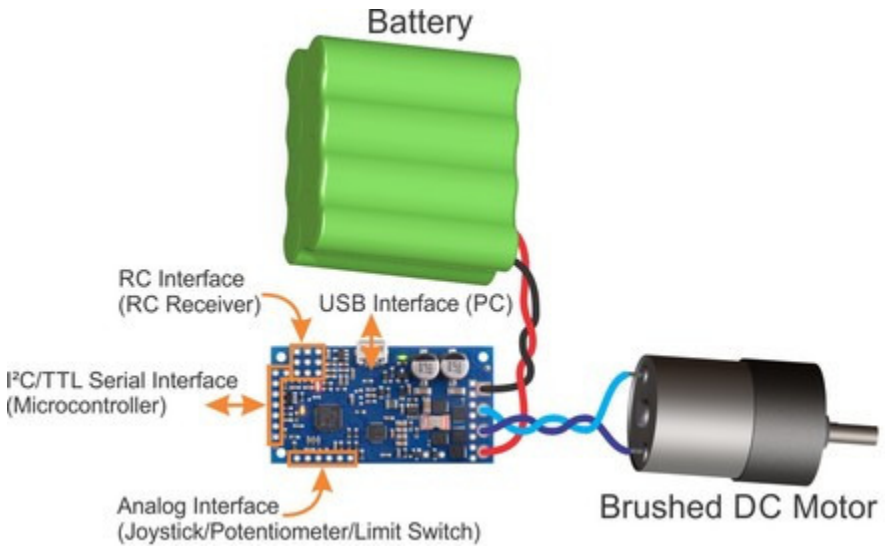
4. Connecting your motor controller

This chapter explains all the electrical connections you might need to make to get your Simple Motor Controller G2 working.

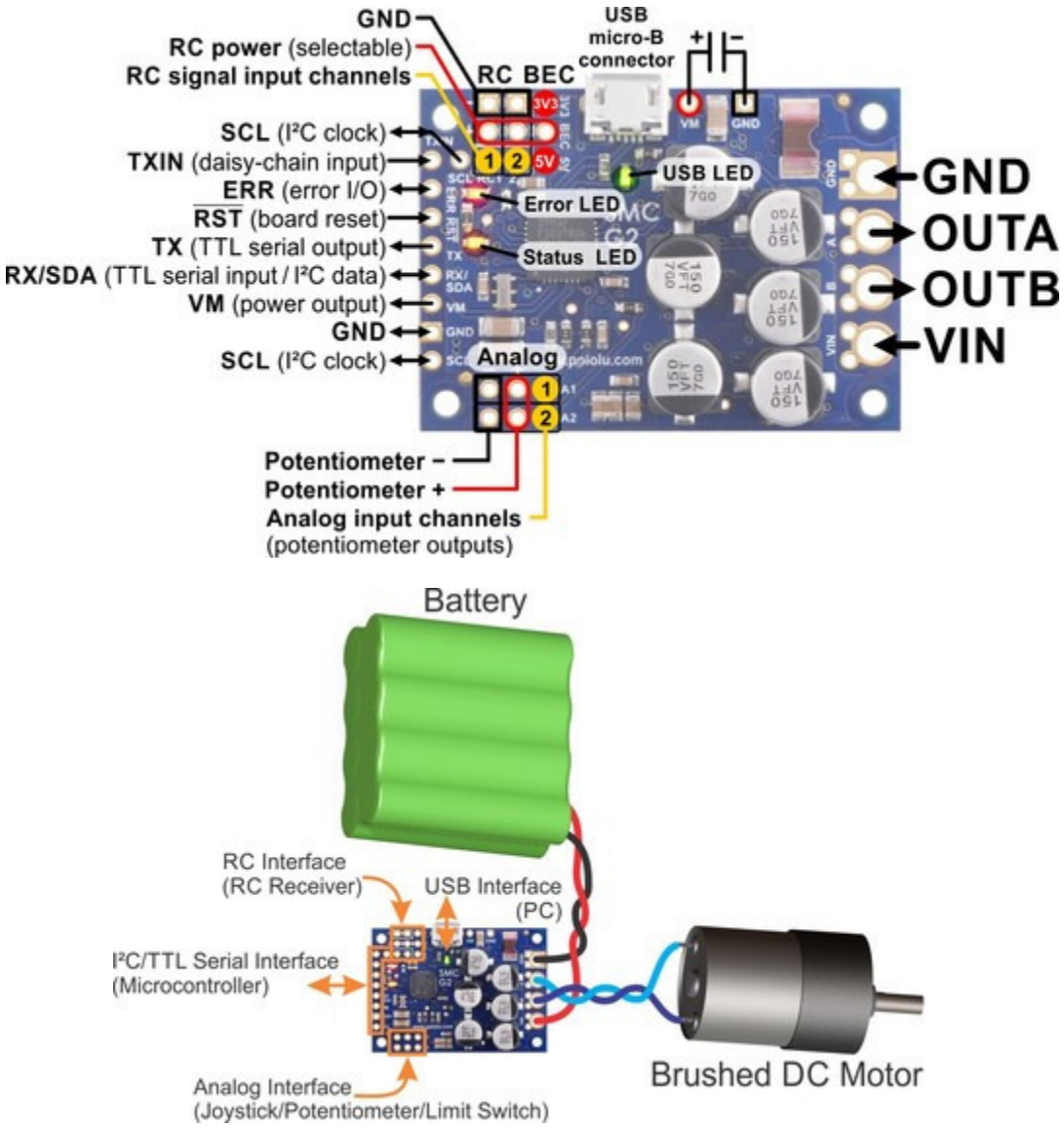
The diagrams below label the key components and pins on the Simple Motor Controllers. Most of these pins are also labeled on the bottom side of the board.

High-Power Simple Motor Controller G2 18v15 and 24v12 pinout





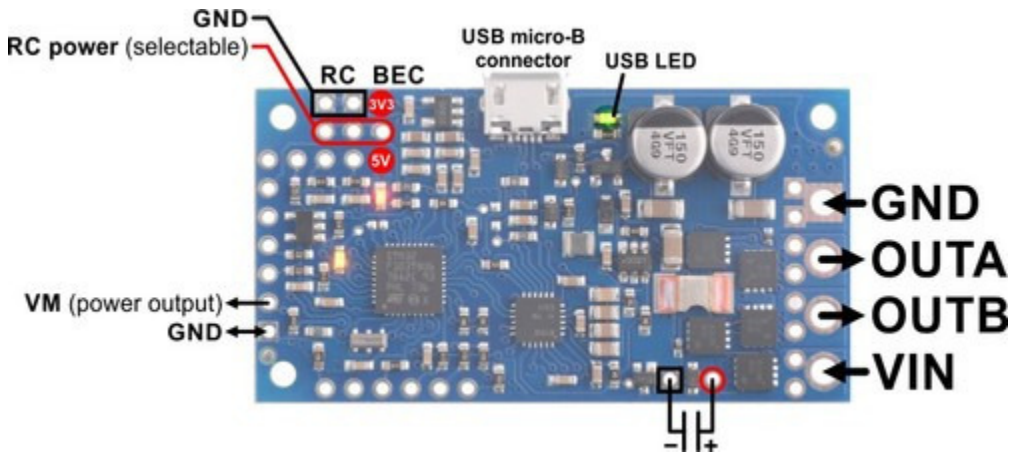
High-Power Simple Motor Controller G2 18v25 and 24v19 pinout



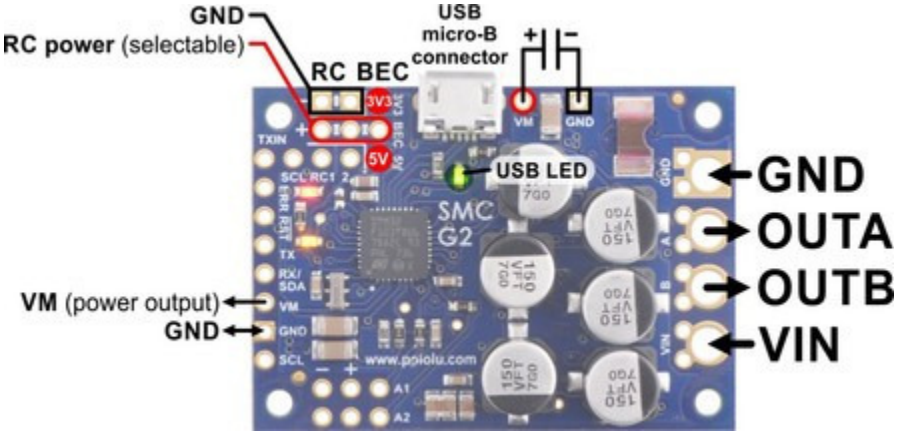
4.1. Connecting power and a motor

> **Warning:** Take proper safety precautions when using high-power electronics. Make sure you know what you are doing when using high voltages or currents! During normal operation, this product can get hot enough to burn you. Take care when handling this product or other components connected to it.

The first step in using your Simple Motor Controller is connecting power and a motor. With those connections in place, you can immediately start testing with the Simple Motor Control Center G2 software. The following section explains the power system in detail.



High-Power Simple Motor Controller G2 18v15 or 24v12 power and motor connections.



High-Power Simple Motor Controller G2 18v25 or 24v19 power and motor connections.

Power considerations

The G2 Simple Motor Controllers can be powered either from USB using a **USB A to Micro-B cable** **[https://www.pololu.com/product/2073]** or from a power supply, such as a battery pack, connected to the large **VIN** and **GND** pads. When the VIN supply is not present, the controller can use USB power to perform all of its functions except for driving the motor. The controller automatically selects VIN as the power source when it is present, even when USB is connected. It is OK to have both USB and VIN power simultaneously connected.

All Simple Motor Controller G2 versions can operate from VIN supplies as low as 6.5 V. The maximum power ratings for the G2 Simple Motor Controllers are shown below:

	18v15	24v12	18v25	24v19
Minimum operating voltage:	6.5 V	6.5 V	6.5 V	6.5 V
Recommended max operating voltage:	24 V[1]	34 V[2]	24 V[1]	34 V[2]
Max nominal battery voltage:	18 V	28 V	18 V	28 V
Max continuous current (no additional cooling):	15 A	12 A	25 A	19 A

1 30 V absolute max.
2 40 V absolute max.

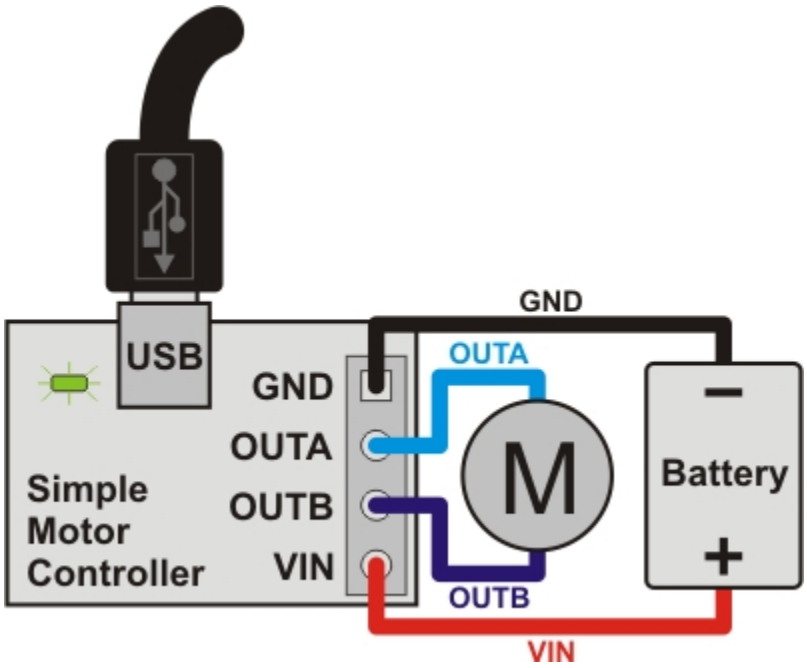
It is very important that you select a power source that does not exceed the absolute maximum voltage rating for your controller. Ripple voltage on the supply line can raise the maximum voltage to more than the average or intended voltage, so we recommend you to select a voltage that leaves at least a 6 V margin for noise. It is also important to note that batteries can be much higher than their nominal voltage when fully charged, so we do not recommend using the 18v15 or 18v25 versions with 24 V batteries unless appropriate measures are taken to limit the peak voltage.

> For 24 V applications, we recommend the 24v12 or 24v19 versions. We strongly recommend against using the 18v7, 18v15, or 18v25 with 24 V batteries, which can significantly exceed 24 V when fully charged and are dangerously close to the maximum voltage limits of these lower-voltage controllers. Using a 24 V battery with an 18vX Simple Motor Controller makes the device much more susceptible to damage from power supply noise or LC voltage spikes.

Finally, make sure you select a power source that is capable of delivering the current your motor will require (e.g. alkaline cells are typically poor choices for high-current applications).

The Simple Motor Controller G2 features a configurable low-voltage shutoff that can help you avoid damaging batteries that are sensitive to over-discharging, such as Li-Po packs. See **Section 5.3** for more information.

Motor considerations



The two terminals of your brushed, DC motor connect to the **OUTA** and **OUTB** pins (which are just labeled "A" and "B" on some boards). When selecting a motor for your controller (or a controller version for your motor), it is important to consider how the motor will be used in your system. If the motor is likely to be stalled for prolonged periods of time or under heavy load, or if the motor will be rapidly changing direction without acceleration limiting enabled, you should be taking into account the stall current of the motor at the voltage it will be running and selecting a controller that can deliver a continuous current that exceeds the stall current.

It is not unusual for the stall current of a motor to be an order of magnitude (10×) higher than its free-run current. When a motor is supplied with full power from rest, it briefly draws the full stall current, and it draws nearly twice the stall current if abruptly switched from full speed in one direction to full speed in the other direction.

Occasionally, electrical noise from a motor can interfere with the rest of the system. This can depend on a number of factors, including the power supply, system wiring, and the quality of the motor. If you notice parts of your system behaving strangely when the motor is active (e.g. corrupted serial data, bad RC pulses, noisy analog voltage readings, or the motor controller randomly resetting), consider taking the following steps to decrease the impact of motor-induced electrical noise on the rest of your system:

1. Solder a **0.1 µF ceramic capacitor** [https://www.pololu.com/product/1166] across the terminals of your motor, or solder one capacitor from each terminal to the motor case. For the greatest

noise suppression, you can use three capacitors (one across the terminals and one from each terminal to the case).

2. Make your motor leads as thick and as short as possible, and twist them around each other. It is also beneficial to do this with your power supply leads.
3. Route your motor and power leads away from your logic connections if possible.
4. Place decoupling capacitors (also known as "bypass capacitors") across power and ground near any electronics you want to isolate from noise.
5. Add a capacitor across the GND and VM pins that are marked with a capacitor symbol in the diagrams at the top of this section.

Power and motor connectors

High-Power Simple Motor Controller G2 18v15 with connectors soldered.

High-Power Simple Motor Controller G2 24v12 with connectors soldered.

The fully-assembled 18v15, and 24v12 controller versions ship with terminal blocks soldered into the large VIN, OUTA, OUTB, and GND pads, as shown in the pictures above. These terminal blocks make it easy to connect and disconnect wires, but they are only rated for 15 A.

High-Power Simple Motor Controller G2 18v15 with included hardware.

High-Power Simple Motor Controller G2 24v12 with included hardware.

High-Power Simple Motor Controller G2 18v25 with included hardware.

High-Power Simple Motor Controller G2 24v19 with included hardware.

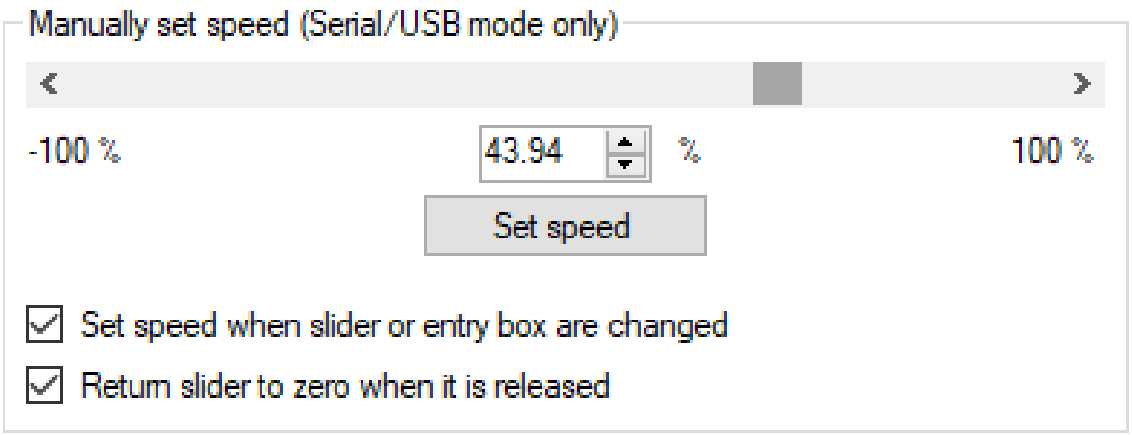
All other versions ship with terminal blocks and male header pins included but not installed, which provides flexibility in making connections. These versions offer two options for connecting to the high-power signals (VIN, OUTA, OUTB, GND): large holes with 5 mm center-to-center spacing, which are compatible with the included **terminal blocks** **[https://www.pololu.com/product/2440]**, and pairs of 0.1"-spaced holes, which are compatible with the included header pins and can be used with perfboards, **breadboards** **[https://www.pololu.com/category/28/solderless-breadboards]**, and 0.1" connectors. For high-power applications that exceed the 15 A rating of the terminal blocks, we recommend soldering thick wires directly to a connector-free version of the board and using **higher-current connectors** **[https://www.pololu.com/product/925]**.

Logic power

The Simple Motor Controllers use 3.3 V logic, but all of the controllers' digital inputs are 5V-tolerant, so it can interface directly with 5V systems. The only pins on the board that cannot tolerate 5V are the two analog input channels, **A1** and **A2**. The simple motor controllers incorporate both a 5V regulator and a 3.3V regulator, but the 5V regulator is only used when power is supplied to VIN. Otherwise, the USB 5V bus voltage replaces the output of the 5V regulator. The 5V and 3.3V power buses are available via the RC BEC jumper pads (see the upper-right corners of the power connection diagrams above), and a shorting block can be used to connect the RC power row to the desired voltage rail, thereby powering a connected RC receiver with 3.3 or 5 V. These pins can also be used to supply approximately 150 mA to other components in your system.

Trying out the controller with USB

Once you have a connected a power supply and a motor, you can use the Simple Motor Control Center G2 to make the motor move and test how various settings affect the behavior of the motor (see **Section 5** for more information on configuring the Simple Motor Controller G2). The Simple Motor Controller G2 defaults to "Serial/USB" input mode, which lets you control the motor speed with the slider bar under the status tab. If you have already changed the input mode of the device to something else, you can restore it by going to the "Input settings" tab, selecting Serial/USB as the input mode, and clicking the "Apply settings" button in the lower right corner.



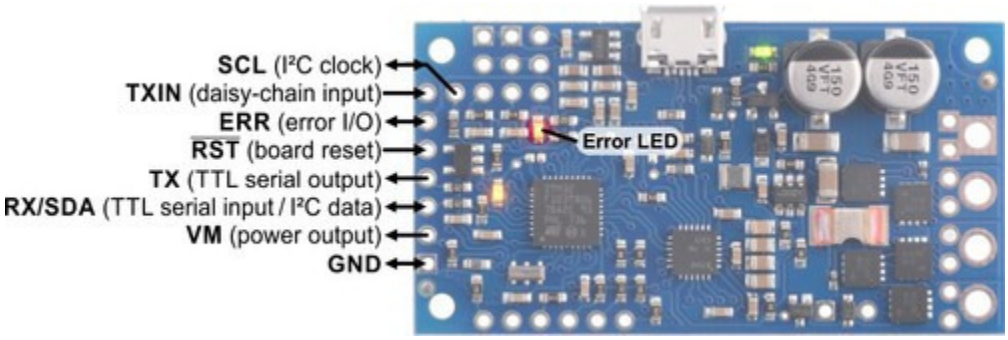
The "Manually set speed" box in the Status tab of the Simple Motor Control Center G2.

Before you can move the motor, you will probably need to click the green Resume button in the lower left corner to clear the safe-start violation. If the Resume button is grayed out, there are errors that are preventing the motor from moving. See **Section 3.3** for information on how to identify and fix errors.

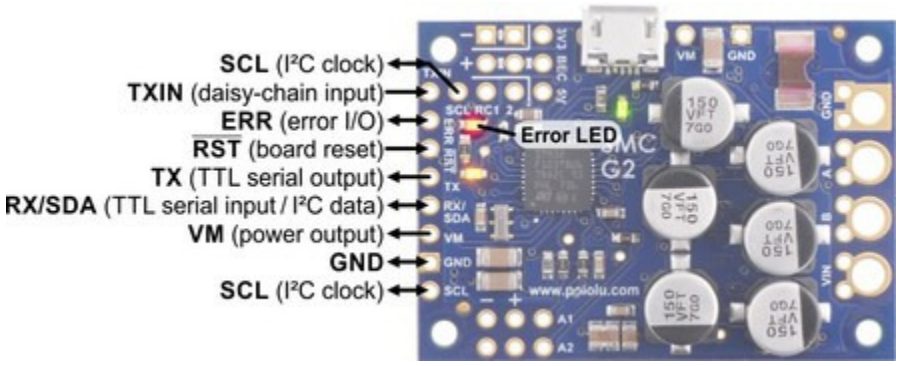
Safe Start is an optional feature, enabled by default, that makes it less likely that the motor will start moving unexpectedly.

4.2. Serial/I²C interface pins

The 0.1"-spaced pins along the left side of the board make it possible to connect the Simple Motor Controller G2 to a microcontroller (e.g. an **A-Star** [https://www.pololu.com/category/149/a-star-programmable-controllers], **Orangutan Robot Controller** [https://www.pololu.com/category/8/robot-controllers], or **Arduino** [https://www.pololu.com/product/2191]) or other logic-level serial I²C device, allowing for the creation of autonomous, self-contained systems. This section explains each of these pins in detail. See **Section 6** for information on the Simple Motor Controller's serial/I²C settings and protocols. See **Section 4.3** for serial wiring examples and see **Section 4.4** for I²C connection examples.



High-Power Simple Motor Controller G2 18v15 or 24v12 serial and I²C connections.



High-Power Simple Motor Controller G2 18v25 or 24v19 serial and I²C connections.

The **GND** pin is a ground connection point. It is electrically connected to the large GND holes on the right side of the board, and the pins labeled with a minus sign in the RC section. Your serial or I²C device must share a common ground with the Simple Motor Controller G2.

The **VM** pin provides access to the board's reverse-protected motor power. The voltage on this pin is not regulated; it will generally be equal to VIN or a little bit lower. This pin is not designed to provide a large amount of current.

The **RX/SDA** pin acts as RX, the TTL serial receive pin, by default. This pin can be connected to the TTL serial output (transmit line) of another device. When I²C is enabled, this pin acts as SDA: an input and open-drain output for sending data. In serial mode, this pin has an internal pull-up resistor enabled, but in I²C mode it does not have any pull-up or pull-down resistors. The RX/SDA pin is 5V-tolerant and protected by a 220Ω series resistor.

The **TX** pin acts as the TTL serial transmit pin. This pin can be connected to the TTL serial input (receive line) of another device. This connection is only required if you want to read information from the motor controller via serial. The TX pin is always an output, and it uses 0 V and 3.3 V voltage levels.

The TX pin is protected by a 220Ω series resistor.

The **SCL** pin acts as the I²C clock signal. The Simple Motor Controller G2 uses a feature of I²C called clock stretching, which means that it will briefly drive this line low while handling I²C transfers. When I²C is disabled, which is the default, this pin acts as a secondary TX pin that ignores the TXIN input. This pin does not have any pull-up or pull-down resistors. The SCL pin is 5V-tolerant and protected by a 220Ω series resistor. The 18v25 and 24v19 versions have two duplicate SCL pins.

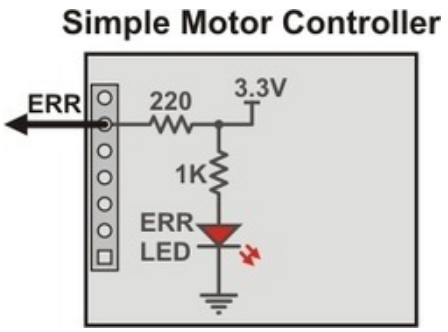
The **$\overline{\text{RST}}$** pin is an active-low reset pin. This pin is internally pulled high; driving it low resets the motor controller. You must wait for at least 1 ms after a reset before transmitting to the controller. This pin can be left disconnected in most applications. This pin is **not** 5V-tolerant.

The **ERR** pin outputs high (3.3 V) when there is an error that is stopping the motor, turning on the red error LED in the process. Otherwise, it is weakly pulled low.

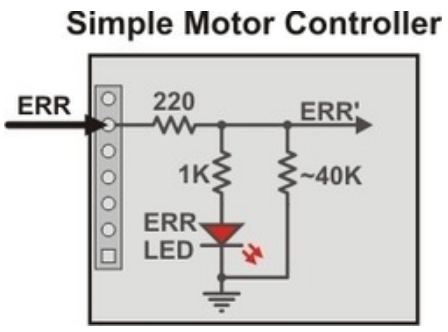
Because the ERR pin never drives low, it is safe to connect the ERR pins of multiple Simple Motor Controllers to the same microcontroller input. If any one of those controllers experiences an error, the microcontroller error input goes high and the error LEDs of all connected Simple Motor Controllers light up.

By default, the ERR pin is also configured to serve as an input that stops the motor when externally driven above 2.3 V. This means that the error lines of multiple controllers can be connected together and all controllers will shut down their motors when any one controller experiences an error. This technique of connecting error lines can be used even when RC signals or analog voltages are used to control the motors.

The following diagrams show the internal circuitry of the ERR pin in the error case (driving high to report an error) and in the error-free case (pulled low and configured as an input):



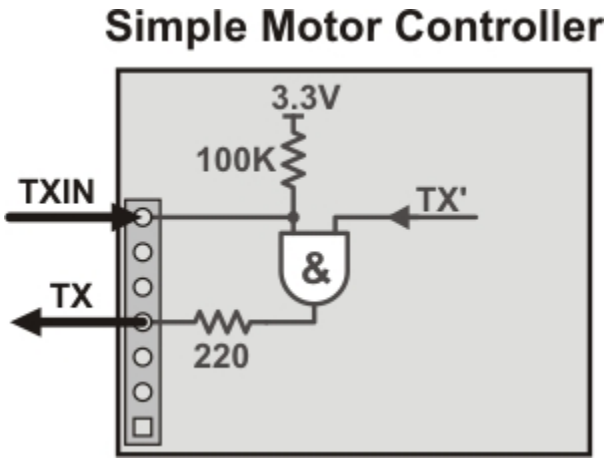
Schematic diagram of the Simple Motor Controller ERR pin when the pin is an output (i.e. there are errors).



Schematic diagram of the Simple Motor Controller ERR pin when the pin is an input (i.e. there are no errors).

The **TXIN** pin is a special input that allows multiple G2 Simple Motor Controllers to be chained together

without requiring an external AND gate. Connecting the transmit output of another serial device to this pin will cause that device's transmissions to be output from the Simple Motor Controller G2's TX pin. Inside each Simple Motor Controller, an AND gate is used to combine the input from the TXIN pin with the controller's own serial transmissions. As long as only one device is transmitting at a time, the transmissions of all chained devices can be read by a single microcontroller receive line. This pin is 5V tolerant. The following diagram shows the internal circuitry of the TX and TXIN pins:



Schematic diagram of the Simple Motor Controller TXIN and TX pins.

4.3. Connecting a serial device

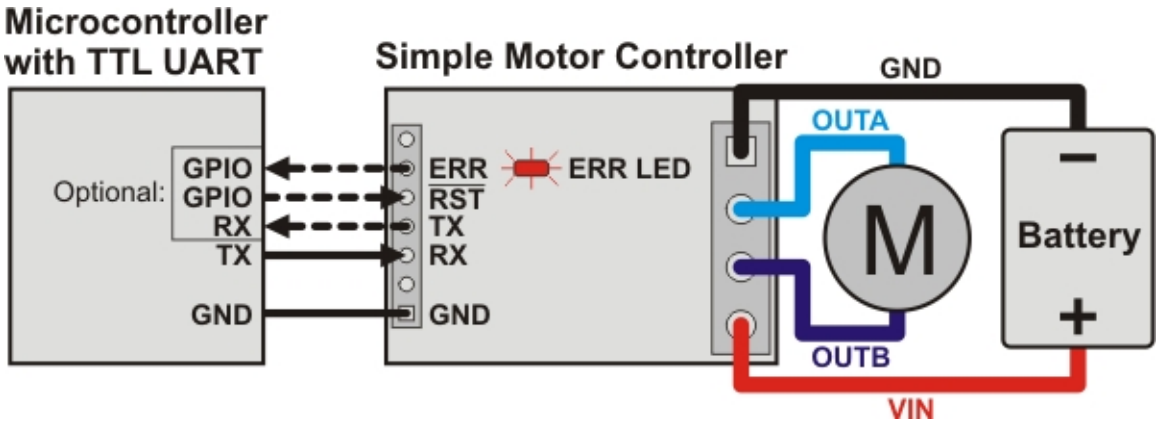
The RX, TX, and TXIN pins on the Simple Motor Controller G2 can be used to communicate with devices with logic-level (TTL) serial interfaces, such as microcontrollers or USB-to-serial adapters. The Simple Motor Controller uses 3.3V logic, but the RX and TXIN inputs 5V-tolerant, which means that the Simple Motor Controller can be used directly with a microcontroller running at 5 V as long as that microcontroller is guaranteed to read the 3.3 V signal from TX as high.

The serial pins use asynchronous, logic-level (TTL), non-inverted serial signals with 8-bit characters and one stop bit (often expressed as 8-N-1). This is the type of serial typically used by microcontroller UART modules.

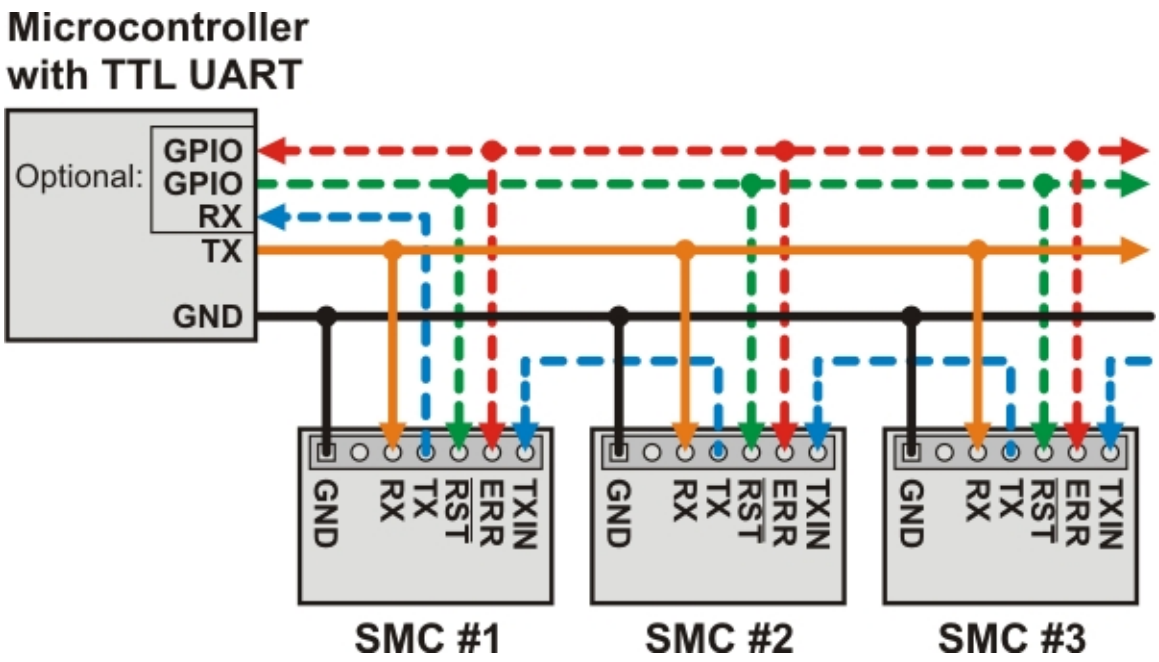
Whenever connecting devices, remember to wire the grounds together, and ensure that each device is properly powered. Unpowered devices with a TTL serial port can turn on or partially on, drawing power from the serial line, which means that extra care must be taken when turning power off and on to reset the devices.

Note: You must use an inverter and level shifter such as a MAX232 or a **Pololu 23201a Serial Adapter [https://www.pololu.com/product/126]** if you want to interface an RS-232 device with the Simple Motor Controller G2. Connecting an RS-232 device directly to the Simple Motor Controller G2 can permanently damage it.

All you need to control the Simple Motor Controller G2 with a microcontroller is a connection between the microcontroller's TTL serial transmit pin and the Simple Motor Controller G2's **RX** pin. If you want to get feedback from the controller, you can connect the **TX** pin to the microcontroller's TTL serial receive pin and/or connect the **ERR** pin to one of the microcontroller's digital inputs. Connecting one of the microcontroller's digital outputs to the **$\overline{\text{RST}}$** pin allows the microcontroller to reset the motor controller.



The following diagram shows how multiple motor controllers can be connected to a single microcontroller UART:



Wiring diagram for controlling multiple Simple Motor Controllers with single TTL serial source, such as a microcontroller.

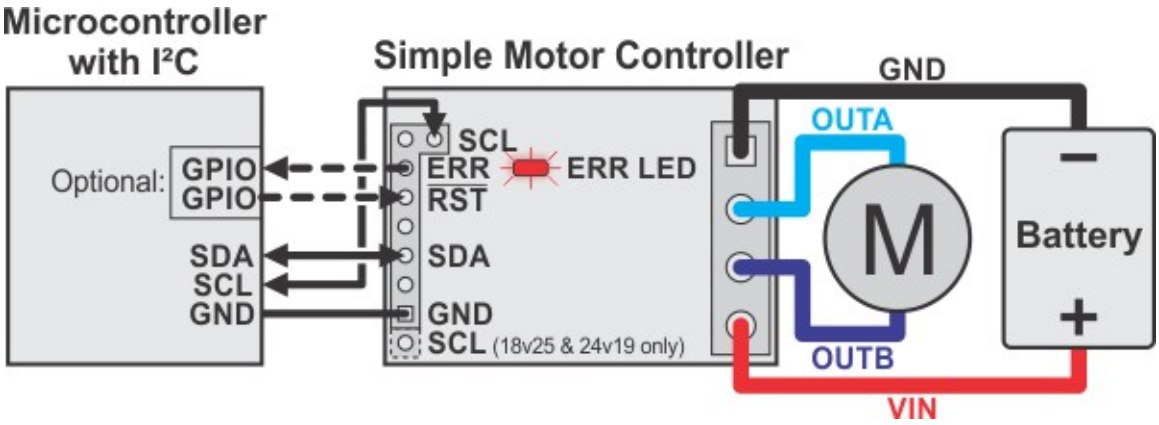
See **Section 6.6** for more information on connecting multiple controllers on the same serial line.

4.4. Connecting an I²C device

The SCL and SDA pins on the Simple Motor Controller G2 can be used to communicate with microcontrollers that support I²C. The Simple Motor Controller G2 acts as an I²C slave and accepts commands from the master device. The Simple Motor Controller G2 is compatible with I²C bus voltage levels from 1.8 V to 5 V.

Note that I²C is not enabled by default. You will need to check the "Enable I2C" checkbox in the "Input settings" tab of the Simple Motor Control Center G2 software, and then click "Apply settings" to enable it.

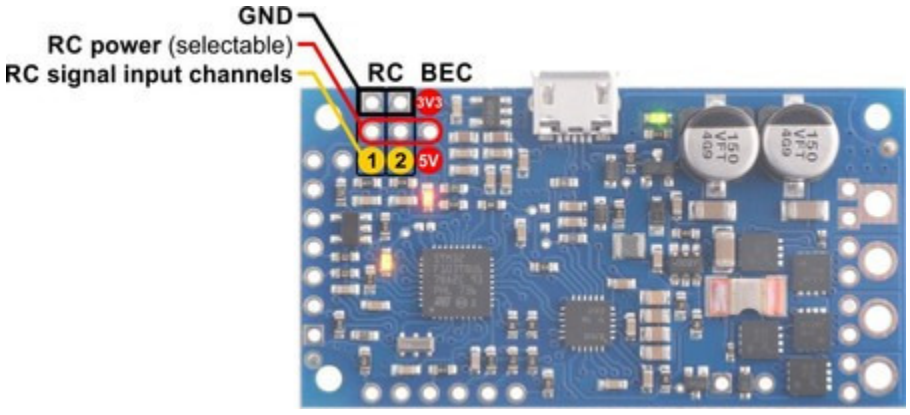
You will need to connect the SCL pin of your master device to the SCL pin of the Simple Motor Controller G2, and connect the SDA pin of the master device to the SDA pin of the Simple Motor Controller G2. (On the 18v25 and 24v19, you can use either of the two SCL pins.) To control multiple devices, connect all the SCL pins together and connect all the SDA pins together.



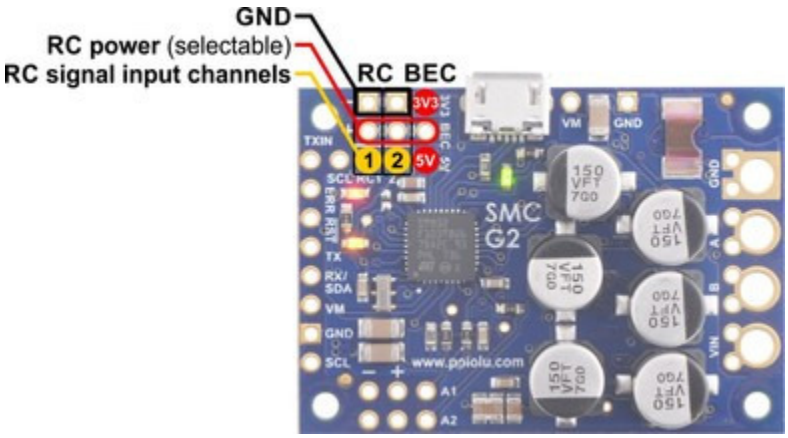
The SDA and SCL pins do not have pull-up resistors enabled, so you should make sure that your master device has pull-ups on both lines, or add external ones yourself. If you find that you need to add pull-up resistors, a resistance of 10 kΩ is a good place to start, but you might need to adjust the resistance depending on how much capacitance is on the bus and how fast you want to run the clock.

4.5. Connecting an RC receiver

The Simple Motor Controller G2 can be directly connected to an RC receiver, allowing for wireless, manual motor control. The RC inputs can serve several functions, from directly controlling the motors (RC input mode) to sending signals to an autonomous robot (Serial/USB mode) to providing an RC kill switch (any input mode). The Simple Motor Controller G2 can derive the motor speed from a single RC input channel, or it can mix the signals on both RC channels to generate the motor speed, which makes intuitive throttle+steering control of a differential-drive robot possible using a pair of controllers. A BEC jumper lets the Simple Motor Controller G2 optionally power your RC receiver at 3.3 V or 5 V, eliminating the need for a second battery.



High-Power Simple Motor Controller G2 18v15 or 24v12 RC connections.



High-Power Simple Motor Controller G2 18v25 or 24v19 RC connections.

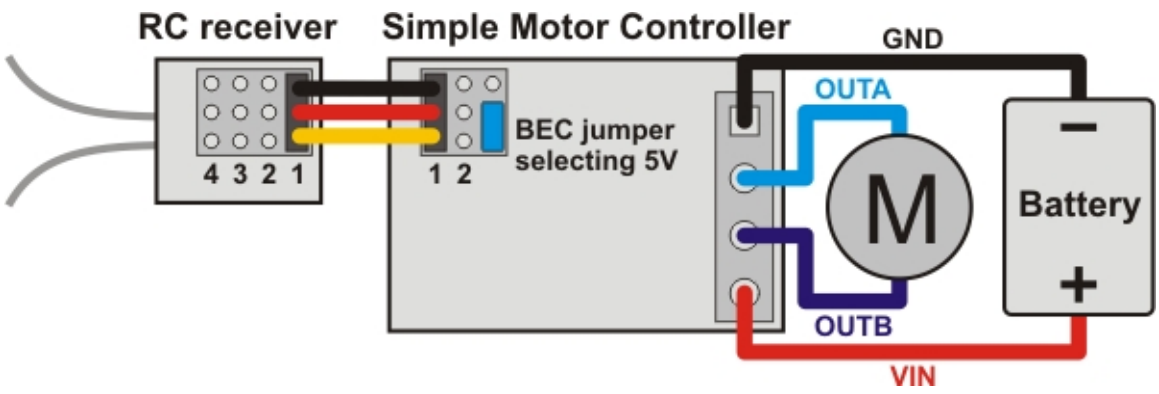
RC connections overview

The RC connection block consists of two channels oriented as columns and a battery elimination circuit (BEC) column for supplying power to the RC receiver. Each channel has a ground pin, a power pin, and a signal pin. The RC signal pins can read standard hobby servo RC pulses with peaks anywhere from 2 V to 5 V. The included **shorting block [https://www.pololu.com/product/968]** can be used to supply the power pin row with either 3.3 V or 5 V, which in turn can be used to power an RC receiver.

> **Note:** If you want to connect servos directly to your RC receiver, you must power it separately as the Simple Motor Controller G2 regulators cannot supply enough current to power a servo. If your RC receiver is powered separately, you must leave the BEC jumper off to avoid shorting the motor controller's regulated voltage to your RC receiver's power source. Your RC receiver and the motor controller must always have a common ground, even if you power the RC receiver separately.

The channel pins have a 0.1" spacing, which means that a **female-female servo extension cable [https://www.pololu.com/product/780]** can be used to connect an RC receiver directly to the board.

Simple wiring example: connecting to an RC Receiver



Wiring diagram for connecting an RC receiver to a Simple Motor Controller.

Using the RC channels

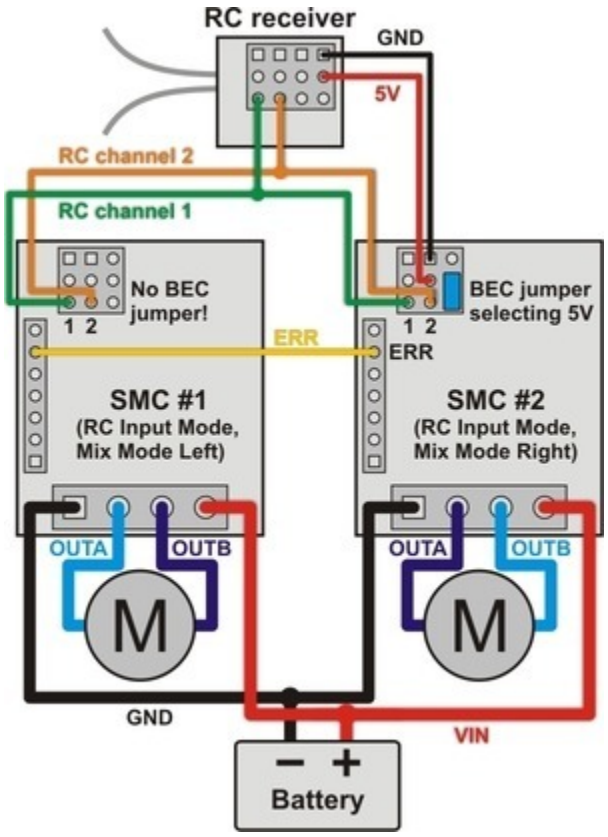
The Simple Motor Controller G2 is constantly reading the two RC channels and making the measured pulse widths available via the USB, serial, and I²C interfaces, even when the controller is not in RC mode. For example, you can use the serial interface to read the RC channel values while the motor controller is in analog mode. The RC channels are read with 0.25 µs resolution, and RC pulse frequencies from 10 Hz to 333 Hz are permitted. The controller has a number of advanced settings for adjusting what constitutes a valid RC signal.

Driving a motor

In RC mode, the channel values are mapped to a motor speed based on the channel calibration values and the mixing mode. We recommend your first step after connecting your RC receiver be to use the Quick Input Setup Wizard in the Simple Motor Control Center G2. The wizard instructs you to move your transmitter control sticks to their extremes and maps stick full forward/right to the maximum forward motor speed, the neutral stick to speed zero, and the stick full back/left to maximum reverse speed. Calibration can have a significant impact on performance.

If mixing mode is disabled, only channel 1 affects motor speed. If mixing mode is set to "right" or "left", channel 1 is considered the "throttle" input and channel 2 is considered the "steering" input. Left mixing mode obtains the motor speed by summing the throttle and steering channels (CH1+CH2) while right mixing mode obtains motor speed by taking the difference of the throttle and steering channels (CH1−CH2). To see why this makes sense, consider a differential-drive robot (a robot with a motor on each side) with a left motor driven by a Simple Motor Controller in left mixing mode and a right motor driven by a Simple Motor Controller in right mixing mode. When throttle is full forward (CH1=max) and steering is neutral (CH2=0), left- and right-mixed motors are both driven forward at full speed and the robot goes forward. When throttle is neutral (CH1=0) and steering is full right (CH2=max), the left mixing results in motor forward at full speed while right mixing results in motor reverse at full speed, so the robot turns right.

As demonstrated above, using both RC channels in mixing mode makes it possible to combine two RC-controlled G2 Simple Motor Controllers to achieve single-stick (mixed) control of a differential drive robot. The following diagram shows how to connect two such motor controllers together:



Wiring diagram for pairing two Simple Motor Controllers with RC channel mixing.

You should configure the controller that drives the right motor as "mixing mode right" and the controller that drives the left motor as "mixing mode left". You can splice together your own cables or use premade **Y splitter cables** **[https://www.pololu.com/product/2164]** to connect channels 1 and 2 from your RC receiver to channels 1 and 2 of both controllers as shown in the diagram above. You can also connect the ERR lines of both controllers together to ensure that both controllers stop if either controller experiences an error.

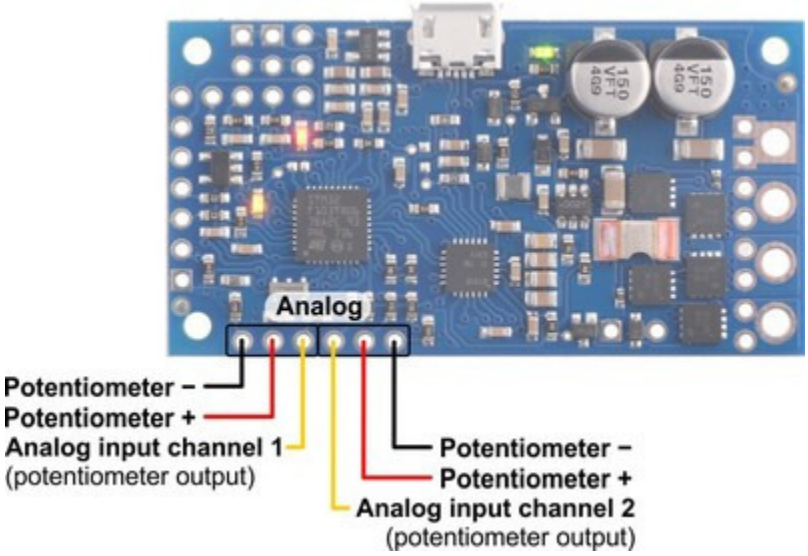
Limit/kill switches

Unused RC channels can also be used as limit or kill switches. For example, you could use an RC signal as a kill switch to stop your autonomous, serially-controlled robot if it gets into trouble. When configured as a limit or kill switch, if the channel's value exceeds more than half of its "forward" value, the switch is activated. We recommend you use the Channel Setup Wizard (click the "Learn…" button

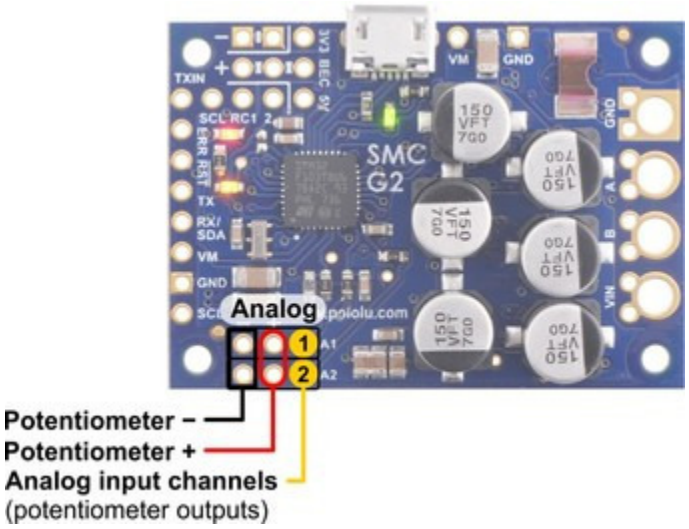
in the Simple Motor Control Center) for any RC channel you configure as a limit or kill switch.

4.6. Connecting a potentiometer or analog joystick

Simple Motor Controller G2 can be directly connected to a 0 V to 3.3 V analog voltage source, such as a potentiometer or analog joystick, allowing for simple manual motor control (e.g. easily control motor speed with a knob). The analog inputs can serve several functions, from directly controlling the motors (Analog input mode) to sending signals to an autonomous robot (Serial/USB mode) to providing limit or kill switch inputs (any input mode). The Simple Motor Controller G2 can derive the motor speed from a single analog input channel, or it can mix the signals on both analog channels to generate the motor speed, which makes intuitive throttle+steering control of a differential-drive robot possible using a pair of controllers. Typical analog voltage sources can be powered directly from the Simple Motor Controller G2.



High-Power Simple Motor Controller G2 18v15 or 24v12 analog connections.



High-Power Simple Motor Controller G2 18v25 or 24v19 analog connections.

Analog connections overview

The analog connection block consists of two channels. Each channel has a signal pin and a **+** and **−** pin for powering the analog voltage source. These potentiometer power pins are special in that they allow the Simple Motor Controller to detect if an analog channel has become disconnected, so we recommend using these pins rather than alternate power supplies or other pins on the board.

If you use an analog voltage source that is not powered from the Simple Motor Controller's potentiometer power (**+** and **−**) pins, you will need to check the "Ignore pot disconnect" checkbox under the "Advanced settings" tab of the Simple Motor Control Center (see **Section 5.3**).

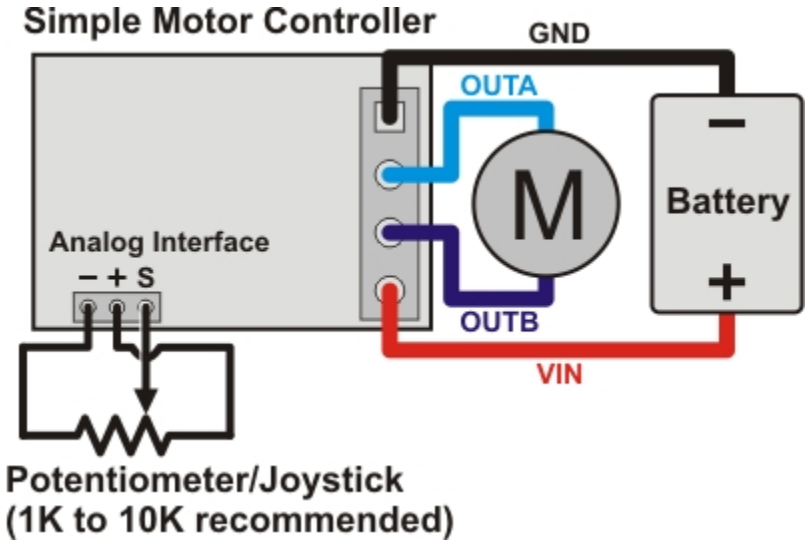
We recommend using a potentiometer in the 1 kΩ to 10 kΩ range. Higher-resistance potentiometers will not work well with the potentiometer disconnection detection feature. If you need to use a higher-resistance potentiometer, you can disable potentiometer disconnection detection from the Simple Motor Control Center G2.

Note: The analog channel inputs are not 5V-tolerant, so you must not connect voltages over 3.3 V to these pins. If your control source outputs voltages higher than 3.3 V, you can use a voltage divider to ensure the voltage is always at an acceptable level.

The channel pins have a 0.1" spacing, which means that a **female-female servo extension cable**

[https://www.pololu.com/product/780] can be used to connect a potentiometer or analog joystick to the controller.

Simple wiring example: connecting to a potentiometer



Wiring diagram for connecting a potentiometer or joystick to a Simple Motor Controller.

Using the analog channels

The Simple Motor Controller G2 is constantly sampling the two analog channels and making the measured voltages available via the USB and serial interfaces, even when the controller is not in analog mode. For example, you can use the serial interface to read the analog channel values while the motor controller is in RC mode. The analog channels are read with 12-bit (0.8 mV) resolution.

Driving a motor

In analog mode, the channel values are mapped to motor speed based the channel calibration values and the mixing mode. We recommend your first step after connecting your analog voltage source be to use Quick Input Setup Wizard in the Simple Motor Control Center G2. The wizard instructs you to move your inputs to their extremes and maps one extreme to the maximum forward motor speed, the neutral position to speed zero, and the other extreme to maximum reverse speed. Calibration can have a significant impact on performance.

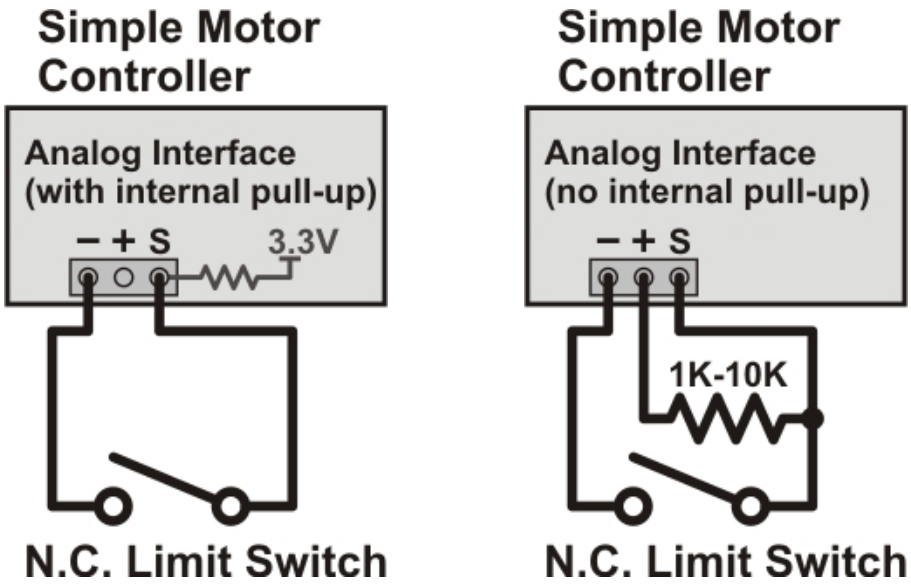
If mixing mode is disabled, only channel 1 affects motor speed. If mixing mode is set to "right" or "left", channel 1 is considered the "throttle" input and channel 2 is considered the "steering" input. Left mixing mode obtains motor speed by summing the throttle and steering channels (CH1+CH2) while right mixing mode obtains motor speed by taking the difference of the throttle and steering channels

(CH1-CH2). To see why this makes sense, consider a differential-drive robot (a robot with a motor on each side) with a left motor driven by a Simple Motor Controller in left mixing mode and a right motor driven by a Simple Motor Controller in right mixing mode. When throttle is full forward (CH1=max) and steering is neutral (CH2=0), left- and right-mixed motors are both driven forward at full speed and the robot goes forward. When throttle is neutral (CH1=0) and steering is full right (CH2=max), the left mixing results in motor forward at full speed while right mixing results in motor reverse at full speed, so the robot turns right.

As demonstrated above, using both analog channels in mixing mode makes it possible to combine two joystick-controlled G2 Simple Motor Controllers to achieve single-stick (mixed) control of a differential drive robot. The connection diagram for such a setup would be very similar to the RC-mixing diagram shown in **Section 4.5**.

Limit/kill switches

Unused analog channels can also be used as limit or kill switches. When configured as a limit or kill switch, if the channel value exceeds more than half of its "forward" value, the switch is activated. If you want to use a push-button switch for this purpose, we recommend using a normally closed (NC) switch connected in one of the two ways depicted in the diagrams below:



By using a normally closed limit switch, you ensure that if the switch becomes disconnected in some way, the controller considers the limit/kill switch active and stops the motor. The left wiring diagram is simpler because it uses an internal pull-up resistor (enabled using the Simple Motor Control Center), but it can only result in one of two possible states: switch active or switch inactive. The right wiring diagram above is able to take advantage of the potentiometer disconnection detection feature. Pressing the switch activates it, releasing it deactivates it, and disconnecting it results in a

disconnection error or an activated switch, depending on which parts of the switch are disconnected.

The above configurations should work with the default analog channel calibration values, but we still recommend you use the Channel Setup Wizard (click the "Learn…" button in the Simple Motor Control Center) for any analog channel you configure as a limit or kill switch.

Normally open (NO) switches can also be used as limit/kill switches with this controller, but they are not as safe since accidental disconnection will lock the switch in an inactive state.

5. Configuring your motor controller

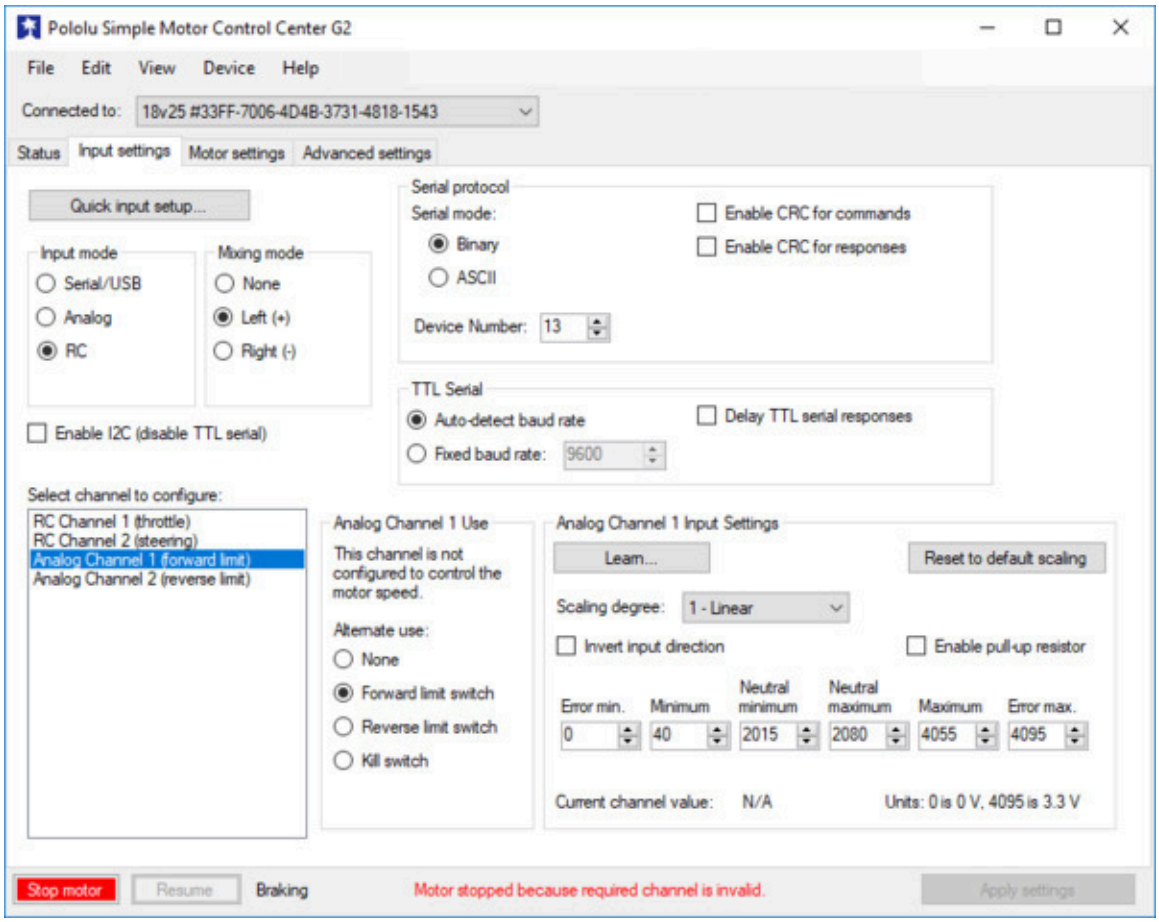
5.1. Input settings

The "Input settings" tab of the Simple Motor Control Center G2 allows you to quickly specify how you want to control the speed of the motor, and also allows you to set up limit and kill switches.

As a first step, we recommend that you click "Quick input setup…". This will launch the Quick Input Setup Wizard, which will let you specify how you want to control of the motor, and (if you are using analog or RC) lets you quickly calibrate your inputs. When you finish the Quick Input Setup Wizard, your new settings will get saved in the "Input settings" tab and will (optionally) be applied to the device so you can start using your new settings right away. After you are done running the wizard, you should be able to successfully control your motor, as long as you have made all the necessary electrical connections as described in **Section 4**.

The rest of this section documents all of the input settings in detail. If you are able to control the motor the way you want to after running the Quick Input Setup Wizard, then you probably don't need to read this section.

The serial and I²C settings in the "Input settings" tab are not documented here. If you want to use serial or I²C, please see **Section 6**.



Input settings tab in the Pololu Simple Motor Control Center G2.

Input mode

The **Input mode** specifies what kind of input the controller will use to calculate the target speed of the motor. The available options are:

- **Serial/USB:** In this input mode, the target speed is specified by serial, I²C, or USB commands, and the target speed is reset to zero whenever there is an error. This is the default input mode.
- **Analog:** In this input mode, the target speed is determined by the voltages measured on the analog signal lines (A1 and optionally A2 if you want to use mixing).
- **RC:** In this input mode, the target speed is determined by the pulse widths measured on the RC signal lines (RC1 and optionally RC2 if you want to use mixing).

Regardless of which input mode you choose, the analog and RC input channels will always be measured; those channels can be used as limit or kill switches if they are not controlling the speed of

the motor and their values can be retrieved using serial, I²C, or USB.

Mixing mode

If you have chosen analog or RC as the input mode, the **Mixing mode** setting specifies whether to use mixing and what type of mixing to use.

The primary use of mixing is for controlling a motor on a differential drive robot. You can use one G2 Simple Motor Controller for each motor on the robot, and feed the same inputs into both of them. We recommend connecting the throttle (forward/reverse) input to channel 1, and the steering (left/right) input to channel 2.

- **None:** In this mixing mode, the target speed is calculated as a function of the scaled value of the first channel only (analog Channel 1 or RC Channel 1).
- **Left (+):** In this mixing mode, the target speed is calculated as a function of the **sum** of the scaled values of both channels.
- **Right (−):** In this mixing mode, the target speed is calculated as a function of the **difference** of the scaled value of both channels (channel 1 minus channel 2).

Note that in RC and analog mode, the target speed depends not only on the scaled values of the channels, but also on the starting speed and max speed parameters, as explained in **Section 5.2**.

The table below summarizes all the input and mixing modes you can choose:

Input mode	Mixing mode	Motor speed is calculated from…	Example Applications
Serial/ USB	N/A	Serial, I²C, and/or USB commands	Motor controlled by microcontroller or PC.
Analog	None	Analog channel 1	Motor controlled by joystick.
Analog	Left (+)	Analog channel 1 plus analog channel 2	Differential drive vehicle controlled by joystick.
Analog	Right (-)	Analog channel 1 minus analog channel 2	Differential drive vehicle controlled by joystick.
RC	None	RC channel 1	Electronic Speed Controller (ESC).
RC	Left (+)	RC channel 1 plus RC channel 2	Differential drive RC vehicle.
RC	Right (-)	RC channel 1 minus RC channel 2	Differential drive RC vehicle.

The settings on the bottom half of the "Input settings" tab are all *channel-specific* settings. To view or edit them, you must first select the desired channel using the list box in the bottom left corner.

Alternate use

The **Alternate use** setting allows you to configure any channel that is not used to control the speed of the motor as a limit or kill switch. The available options are:

- **None:** This channel will not be used for anything special, but its raw and scaled values can be read using serial or USB.
- **Forward limit switch:** When the scaled value of the channel is above 1600 (50%), the limit switch will be considered active and the motor will not be allowed to move forward. If the target speed is positive, a "Limit/kill switch" error will occur.
- **Reverse limit switch:** When the scaled value of the channel is above 1600 (50%), the limit switch will be considered active and the motor will not be allowed to move in reverse. If the target speed is negative, a "Limit/kill switch" error will occur.
- **Kill switch:** When the scaled value of the channel is above 1600 (50%), the kill switch will be considered active and the "Limit/kill switch" error will occur, preventing the motor from moving. For example, you could use the kill switch feature and the Serial/USB input mode to make an autonomous robot that you can conveniently immobilize from a distance using an RC transmitter and receiver.

The forward and reverse limit switch options allow you to set up limits that prevent your actuator from moving out of its allowed range. You will probably want to avoid setting a motor deceleration limit if you are using a limit switch, because the deceleration limit will prevent the motor from stopping immediately: when the switch is triggered, the motor will gradually decelerate from its current speed to zero, which might be bad for your system depending on how it is set up. See **Section 4.4** and **Section 4.5** for information about connecting limit switches. See **Section 5.1.1** for more information about configuring a limit or kill switch.

Any channel configured as a limit or kill switch is considered a required channel. This means that the motor will stop if that channel becomes disconnected (the Required channel invalid error will occur).

Learn button

The **Learn…** button launches the Channel Setup Wizard, which lets you quickly calibrate your input channel or limit switch. Before using this wizard, you should select your desired alternate use, and if you are configuring an analog channel then you should first enable the pull-up resistor and/or check "Ignore pot disconnect" in the "Advanced settings" tab if necessary.

Enable pull-up resistor (analog channels only)

When checked, the **Enable pull-up resistor** option enables a pull-up resistor on the selected analog input line. The value of the resistor is approximately 40 kΩ and it pulls the line up to 3.3 V.

Scaling parameters

The rest of channel-specific settings are all scaling parameters, which means they specify how the scaled value of the input channel is calculated from its raw value. They also specify the normal range of the input channel. All of these parameters except **scaling degree** can be easily set using the **Learn…** button.

The raw value of a channel is measured directly from the input pin. For RC channels, the raw value is the width of received pulses in units of 1/4 µs; typical RC receivers will generate signals between 4000 (1000 µs) and 8000 (2000 µs). For analog channels, the raw value is a 12-bit measurement of the voltage on the input line: 0 is 0 V and 4095 is 3.3 V. You can see the raw value of the selected channel by looking at the "Current channel value" label or by looking at the Status tab.

If the raw value is less than "Error min." or greater than "Error max.", then the channel is considered invalid and the scaled value is not computed. Otherwise, the scaled value of a channel is calculated from the raw value using the scaling parameters. Specifically:

- Raw values between **Error min.** and **Minimum** map to a scaled value of −3200 (or 3200 if "Invert input direction" is checked).
- Raw values between **Minimum** and **Neutral minimum** map to a scaled value between

−3200 (or 3200 if "Invert input direction" is checked) and 0.

- Raw values between **Neutral minimum** and **Neutral maximum** map to a scaled value of 0.
- Raw values between **Neutral maximum** and **Maximum** map to a scaled value between 0 and 3200 (or −3200 if "Invert input direction" is checked).
- Raw values between **Maximum** and **Error max.** map to a scaled value of 3200 (or −3200 if "Invert input direction" is checked).

By default, the scaling is linear, but you can change the **Scaling degree** to use a higher-degree polynomial function, which gives you better control for low speeds.

The **Error min.** and **Error max.** parameters should be set so that the input channel's raw value is always within that range whenever the input is operating properly. One way to do this is to move your input to the minimum position, and set **Error min.** to be 10 to 200 counts lower than the current channel value. Similarly, move your input to its maximum position, and set **Error max.** to be 10 to 200 counts higher than the current channel value.

The **Minimum** and **Maximum** parameters should be as far apart as possible to maximize the accuracy of your speed control, but they should still be close enough that you can reliably reach scaled values of ±3200 (±100 %). One way to do this is to move your input to its minimum position, and set **Minimum** to be 10 to 200 counts higher than the current channel value. Similarly, move your input to its maximum position, and set **Maximum** to be 10 to 200 counts lower than the current channel value.

The **Neutral minimum** and **Neutral maximum** parameters should be as close as possible to maximize the accuracy of your speed control, but they should still be far enough from each other so that you can reliably reach a scaled value of 0 when you put your input in the neutral position (e.g. release your finger from the joystick). Some joysticks can settle at different positions depending on where you release the from, so you should experiment with releasing your joystick from different positions and see what raw values you get (you can see them using "Current channel value" label or in the Status tab). Then set **Neutral minimum** and **Neutral maximum** so that their range includes all of the values you saw, and has a reasonable margin. This guarantees that you will not waste any power driving your motor when your stick is in the neutral position.

If you want to restrict the scaled value of the channel to always be negative or always be positive, you can set the **Minimum** equal **Neutral Minimum** or you can set **Maximum** equal to **Neutral Maximum**. This could be useful for one-directional control of a motor but typical applications will not need this.

5.1.1. Configuring a limit or kill switch

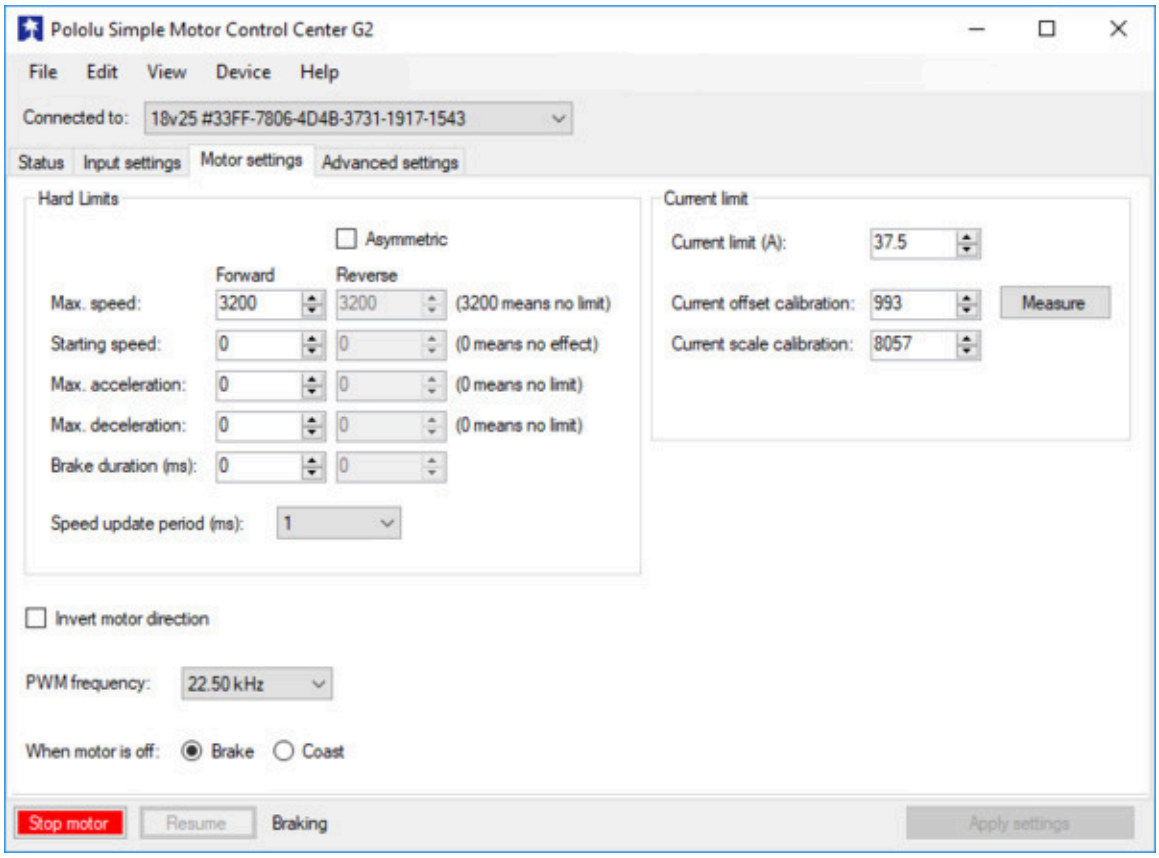
Limit switches and kill switches help protect your motor controller from performing unwanted actions. For example, analog limit switches could be configured to prevent your actuator from moving out of its valid range. An RC kill switch could be configured to conveniently immobilize an autonomous robot from a distance using an RC transmitter and receiver.

To configure your Simple Motor Controller G2 to use a limit or kill switch, follow these steps:

1. Decide what channel you are going to connect your limit switch to, and connect it to that channel as described in **Section 4.5** if it is an RC limit or kill switch or **Section 4.6** if it is an analog limit or kill switch.
2. If you are using an analog channel for your limit switch and you decide to use the internal pull-up instead of supplying an external one, check the "Enable pull-up resistor" box for that channel in the "Input settings" tab.
3. If you are using an analog channel and have chosen a wiring configuration that makes it impossible for the controller to detect when your switch is disconnected, check "Ignore pot disconnect" box in the "Advanced settings" tab. Disconnect detection works by toggling power to the analog power pins (+) and making sure that this toggling has an effect on the voltage on the signal pin (A1 or A2). If you have wired your switch such that the analog power pin is not connected to the signal pin, you will need to do this.
4. Select the desired "Alternate use" for the switch channel. This determines whether it will be a forward limit switch, reverse limit switch, or kill switch. See **Section 5.1** for details about this setting.
5. Click "Apply settings".
6. Look at the current channel value label in the "Input settings" tab. Press or activate your switch and make sure that the channel value changes significantly. If the value does not change, then you should double check your connections and settings and try again.
7. Click the "Learn…" button for the channel in the Input Settings tab. The Channel Setup Wizard will walk you through the steps needed to calibrate your switch's scaling parameters.

5.2. Motor settings

The "Motor settings" tab of the Simple Motor Control Center G2 allows you to set up limits to protect your system and lets you specify the details of how your motor should be driven.



Motor settings tab in the Pololu Simple Motor Control Center G2.

Hard limits

The **Hard limits** box allows you to set up hard limits on the motion of your motor in order to protect your system and reduce mechanical stress.

They are called **Hard Limits** because they are stored in non-volatile memory and they are always obeyed. However, all of them except "Starting speed" can be temporarily modified using the appropriate USB or serial command. Only modifications that make the system safer are allowed. These temporary changes will only last until the next time the device resets, at which point the hard limits will be reloaded. See **Section 6.2.1** for more details about setting temporary motor limits.

If you want to enter different limits for the reverse and forward directions, check the **Asymmetric** checkbox.

Max speed is a number between 0 and 3200 that specifies the maximum speed at which the motor controller will ever drive the motor. The default value is 3200, which corresponds to 100% and means there is no limit. A value of 0 means that the motor will not be allowed to drive in the specified direction. This setting also affects how the target speed is computed in RC and analog modes: after mixing is

optionally performed, a scaled value of 3200 or −3200 maps to the Max speed. The Max speed should be zero or it should be greater than the Starting speed.

Starting speed is a number between 0 and 3200 that specifies the minimum speed at which the motor controller will ever drive the motor. The default value is 0, which means there is no minimum, so this setting has no effect. This setting also affects how the target speed is computed in RC or analog modes: after mixing is optionally performed, a scaled value of 1 means the target speed equals the forward starting speed and a scaled value of -1 means the target speed equals the inverse (negation) of reverse starting speed. The starting speed parameter allows you to save some energy by never driving the motor at speeds that are too low to actually make the motor turn. It can also make your joystick control be more accurate and responsive, because the motor can start moving as soon as the stick leaves the neutral area.

Max. acceleration is a number between 0 and 3200 that specifies how much the magnitude (absolute value) of the motor speed is allowed to increase every speed update period. The default value is 0, which means there is no limit. An acceleration limit can help reduce mechanical stress and help reduce current spikes when the motor is starting up. If an acceleration value of 1 is too fast for your application, you can increase the speed update period to make it slower.

Max. deceleration is a number between 0 and 3200 that specifies how much the magnitude (absolute value) of the motor speed is allowed to decrease every speed update period. The default value is 0, which means there is no limit. A deceleration limit can help reduce mechanical stress and help reduce current spikes when the motor is decelerating. Note that deceleration limits apply even when there is an error stopping the motor; depending on your setup, it might not be a good idea to use deceleration in conjunction with a limit switch because the motor will not stop as fast as possible when the limit switch is triggered. If an deceleration value of 1 is too fast for your application, you can increase the speed update period to make it slower.

Brake duration is the time, in milliseconds, that the motor controller will spend braking the motor (current speed = 0) before allowing the current speed to change signs. The forward brake duration is the braking time required before switching from forward to reverse (from positive to negative speeds). The reverse brake duration is the braking time required before switching from reverse to forward (from negative to positive speeds).

The **Speed update period** is the time, in milliseconds, between consecutive updates to the current speed. The default is 1 ms, which is the lowest allowed value. By increasing the speed update period, you can decrease the effective rate of acceleration and deceleration because the updates will be applied less often. The slowest possible acceleration/deceleration can be achieved by setting the Speed update period to 100 ms and the acceleration/deceleration limit to 1; with this configuration it will take 320 seconds to accelerate from speed 0 to speed 3200 (100 %) or decelerate from speed 3200 to speed 0.

Current limit

The **Current limit** setting sets the hardware current limit for the Simple Motor Controller G2's motor driver; when the motor current exceeds this value, the driver will actively limit it using current chopping. Hardware current limiting is performed entirely by the motor driver, which means it can react quickly to current spikes (within a few microseconds).

The Simple Motor Controller G2 obtains a raw current sense measurement from the motor driver in the form of an analog voltage, which typically has an offset of about 50 mV when VIN is present, though the offset can vary widely from unit to unit. The controller converts this value into a current in units of milliamps with the following formulas (where the raw current sense measurement is V_{raw} and the reported current is I):

$$V_{\text{corrected}} = V_{\text{raw}} - \frac{\textbf{“Current offset calibration”}}{19.859}\text{mV}$$

$$I = \frac{V_{\text{corrected}} \times \textbf{“Current scale calibration”} \times \text{Version-specific conversion factor}}{|\text{Current speed}|}$$

The default value of the **Current offset calibration** is 993, which corresponds to 50 mV. Increasing it makes the reported current lower, while decreasing it makes the reported current higher.

If the controller reports extremely high, inaccurate currents while the motor is driving at a low duty cycle, it might be due to the analog current sense signal from the motor driver having an offset higher than the typical 50 mV offset. You can make the current readings more accurate at low duty cycles by setting the “Current offset calibration” setting properly.

The easiest way to set the “Current offset calibration” setting is to connect power to VIN, make sure the motor is stopped, and then click the “Measure” button to measure the offset and automatically change the setting.

The default value of the **Current scale calibration** setting is 8057. Increasing it makes the reported current higher, while decreasing it makes the reported current lower.

Current limit setting: converting from milliamps to internal units

Internally, the current limit setting has a value between 0 and 3200 and uses internal units. The Simple Motor Control Center G2 software lets you enter the desired current limit in Amps, and converts to the internal units for you. However, if you want to set the current limit using the “Set current limit” serial/I²C command, you will need to do that conversion yourself by following these steps:

1. Take the desired current limit in units of milliamps and multiply it by 3200.

2. Multiply this number by the following number, which depends on the Simple Motor Controller G2 model: 2 for the 18v15, 3 for the 24v12, 1 for the 18v25, or 2 for the 24v19.
3. Divide by the current scale calibration setting (8057 by default).
4. Add the current offset calibration setting (993 by default).
5. Multiply by 3200 and then divide by 65536 to get the final result.

Current limit measurement: converting from a raw measurement to milliamps

The Simple Motor Controller G2 reports the motor current in milliamps, but it also reports a raw current measurement reading. This reading is the voltage on the current sense line: 0 represents 0 V, while 65536 represents 3300 mV. If you want to do the calculation to convert the raw reading to milliamps yourself, follow these steps:

1. Take the raw current measurement and subtract the current offset calibration setting (993 by default) from it. If this yields a negative number, the current is 0 mA and you can skip the steps after this one.
2. Multiply the result by the current scale calibration setting (8057 by default).
3. Divide by the following number, which depends on the Simple Motor Controller G2 model: 2 for the 18v15, 3 for the 24v12, 1 for the 18v25, or 2 for the 24v19.
4. Divide by the absolute value of the current motor speed as a number from 1 to 3200. If the motor speed is 0, treat the current as 0 mA instead.

Other motor settings

The **Invert motor direction** option lets you switch the meanings of forward and reverse. By default, forward means the average voltage on OUTA is greater than the average voltage on OUTB (and reverse means the opposite). With the Invert motor direction option enabled, forward means the average voltage on OUTA is less than the average voltage on OUTB.

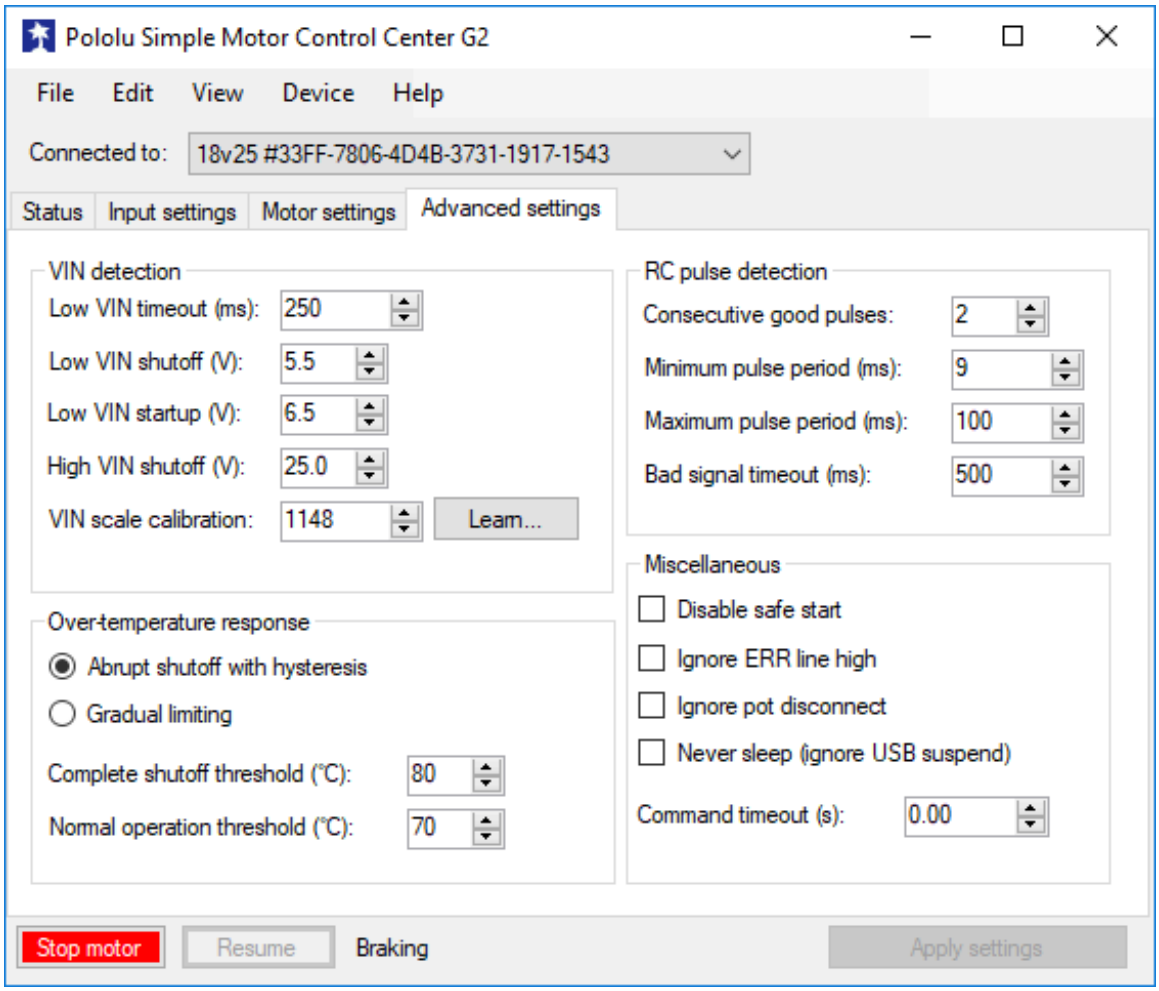
The **PWM frequency** setting specifies the frequency of the rapidly-switching (PWM) signal used to control the speed of the motor. Several PWM frequency options are available between 1.13 and 22.50 kHz. The default PWM frequency is 22.50 kHz. This is an ultrasonic frequency; it is too high for humans to hear, so you won't hear the high-pitched whine from the motor that other motor controllers can cause. Using a lower PWM frequency will reduce switching losses and slightly increase the power output to the motor because the duty cycle (the percentage of the time that the H-bridge is powering the motor) can be closer to 100%.

The **When motor is off** setting specifies the controller's behavior when the current speed is zero (because of an error or any other reason). The default option is **Brake**, which means that the controller

will drive its motor outputs low, the motor will brake, and it will be relatively hard to turn the motor by hand. If you would prefer that your motor stop more gradually or be easier to turn while it is stopped, you can use the **Coast** option, which causes the motor outputs to be disabled instead. Another way to have gradual stops is to set a deceleration limit, which will cause the current speed to slowly drop to zero.

5.3. Advanced settings

The "Advanced settings" tab of the Simple Motor Control Center G2 lets you fine-tune the details of how your Simple Motor Controller G2 behaves.



Advanced settings tab in the Pololu Simple Motor Control Center G2.

VIN detection

The Low VIN options specify what constitutes a Low VIN error. A Low VIN error occurs when the voltage on the VIN line drops below the **Low VIN shutoff** voltage and stays below it for the amount of

time specified by the **Low VIN timeout**. The Low VIN error will stop occurring when the voltage on the VIN line rises above the **Low VIN startup** voltage.

If you are using a battery that can be damaged by over-discharging, we recommend setting your Low VIN shutoff to an appropriate value so that your motors will shut down when the battery voltage gets too low. For example, if you are using a Lithium-Polymer (Li-Po) battery, it would be good to set the Low VIN shutoff to something like 3.0 V or 3.5 V multiplied by the number of cells in your battery. You should also consult your battery's specifications, and adjust your Low VIN shutoff based on how much current your motor draws and how careful you want to be.

If VIN exceeds the **High VIN shutoff** level, a High VIN error will occur. This error is different from the other errors: it instantly shuts down the motors and goes into full braking mode, regardless of your other settings. This means that if you are using the controller in a differential drive vehicle and your vehicle is being pulled down a hill by gravity, the extra voltage generated by the motors might trigger a VIN error and the controller would attempt to stop your robot's descent by braking.

The **VIN multiplier offset** is a calibration factor used in computing VIN. The default value of 0 should work fine for most purposes. If you have a multimeter or another accurate way of measuring voltage, you can click the **Learn…** button to have the software automatically set this number for you. If you find that the VIN reading shown in the Status tab is too high, you should decrease this number. If it is too low, you should increase this number.

Over-temperature response

The Simple Motor Controller G2 monitors the temperature of the board at two points near the MOSFETs and protects itself from burning up by generating an error when the temperature is too high. The Simple Motor Controller G2 has two modes for over-temperature response:

Abrupt shutoff with hysteresis: This is the default mode. In this mode, the Over temperature error will start occurring when either temperature measurement exceeds the "Complete shutoff threshold" and will keep occurring until both temperatures drop below the "Normal operation threshold". In this mode, it will be obvious when you are having temperature issues because your motor will shut down completely while your motor controller cools off.

Gradual limiting: In this mode, whenever the temperature is between the "Normal operation threshold" and "Complete shutoff threshold", the magnitude of the motor speed will be limited. The speed limit is 3200 (100%) when the temperature is equal to the "Normal operation threshold", and it decreases linearly with temperature so that the speed limit is 0 when the temperature is equal to the "Complete shutoff threshold". If the temperature rises above the "Complete shutoff threshold" an over-temperature error will occur and the motor will stop. In this mode, the motor will keep on running as the board heats up, but it might run slower due to the temperature-based speed limiting.

The "Complete shutoff threshold" should be greater than or equal to the "Normal operation threshold".

RC pulse detection

These parameters adjust how lenient or strict the RC signal measurement is on the RC1 and RC2 lines. If you use strict settings, your controller will shut down faster when the RC signal is lost and be less likely to act on corrupted data. If you use lenient settings, your controller will be more likely to keep operating when the RC signal quality is poor.

Consecutive good pulses is the number of consecutive good pulses that must be received before the controller starts heeding good pulses and updating the channel value. The default value of 2 means that after 2 good pulses in a row are received, the third one will be used to update the channel value. A value of 0 means that every good pulse results in an update of the channel value.

Minimum pulse period and **Maximum pulse period** specify limits on the amount of time allowed between pulses. If a pulse is received too soon after a previous pulse, it is considered bad. If the pulses on the line stop, then the RC input channel's signal is considered invalid after an amount of time equal to the **Maximum pulse period** has elapsed. The period of your RC signal is shown in the Status tab, so you can use that to help pick good values for these settings.

The **Bad signal timeout** is like an expiration time for the pulses. If the RC signal line is corrupted by enough bad pulses that the channel's value is not getting updated, then the RC input channel's signal will be considered invalid after an amount of time equal to the **Bad signal timeout** has elapsed.

Miscellaneous

The **Disable safe start** option disables Safe-start violation error, which is described in **Section 3.3**. In Serial/USB input mode, this means that you will no longer have to send Exit Safe Start commands. In RC or Analog input mode, this means that you will no longer have to center your inputs in order to restart the motor after an error. This option makes it more likely that the motor will start when you are not expecting it.

The **Ignore ERR line high** option disables the ERR line high error, which is described in **Section 3.3**. This allows your motor to run even if the ERR line is being driven high by some external device.

The **Ignore pot disconnect** option disables the disconnect detection for analog channels. Enabling this option means that the device will stop toggling the positive (+) analog power pins in order to detect whether your potentiometer is connected. The analog channel will still be considered invalid if the voltage goes out of the acceptable range specified by the "Error min" and "Error max" parameters for that channel. This option is necessary if you are connecting a device to the analog input in a way that prevents the disconnect detection from working.

The **Never sleep (ignore USB suspend)** option prevents the device from going into deep sleep

mode in order to comply with the suspend current requirements of the USB specifications. Checking this option will make the device non-USB compliant, but will allow it to perform some functions while connected to a sleeping PC via USB and the VIN power supply is disconnected. Note that the Simple Motor Controller G2 can not drive a motor while VIN is disconnected.

The "Command timeout" error occurs if you are controlling your motor using a microcontroller or a PC (the input mode is Serial/USB) and the **Command timeout** period has elapsed with no valid serial or USB commands being received by the controller. The default value of **Command Timeout** is 0, which means the error is disabled. The **Command Timeout** can be specified with 0.01 s resolution and can be as high as 655.35 s. The purpose of the "Command timeout" error is to ensure that your motor will stop if the software talking to the controller crashes or if the communications link is broken. For more details about this error see **Section 3.3**.

5.4. Upgrading firmware

The Simple Motor Controller G2 has field-upgradeable firmware that can be easily updated when we release bug fixes or new features. We have not released any firmware upgrades yet.

6. Using the serial and I²C interfaces

This section documents the following three interfaces of the Simple Motor Controller G2:

- The **USB virtual serial port** (COM port), which is part of the USB interface.
- The **TTL serial port**, consisting of the controller's RX and TX pins.
- The **I²C interface**, consisting of the controller's SDA and SCL pins.

Each of these three interfaces support the same binary command protocol, as described in **Section 6.2**.

The TTL serial and USB virtual serial port also support an ASCII protocol as described in **Section 6.3**.

You can use these interfaces to set the speed of the motor when the controller's "Input mode" setting is Serial/USB. In any input mode, you can use these interfaces to request information about the motor controller's state and monitor the RC and analog channel inputs.

The Simple Motor Controller G2 treats each interface independently, so it is OK to use the USB serial port while simultaneously using TTL serial or I²C.

The Simple Motor Controller G2 can also be controlled using its native USB interface (see **Section 7**), which uses its own, separate protocol.

USB virtual serial port

The Simple Motor Controller G2 installs as two devices, one of which is a virtual serial command port. On Windows, you can identify the controller's COM port number by looking in your computer's Device Manager and expanding the "Ports (COM & LPT)" category.

On Linux, the serial port name will be something like `/dev/ttyACM0`.

On macOS, the serial port name will be something like `/dev/cu.usbmodemfa121`.

You can use a terminal program or computer software to send commands to this virtual serial port over USB. Most common programming languages have libraries for sending serial data (e.g. Visual C# has a SerialPort class), which makes it easy to write a custom computer program to control the Simple Motor Controller G2. See **Section 8** for code examples. The baud rate settings do not matter when communicating through the virtual COM port.

TTL serial port

The controller's serial receive line, **RX**, can receive bytes from a TTL serial source, such as a microcontroller, which allows for integration into embedded systems. The RX line expects a logic-level

(0 to 2–5 V, or "TTL"), non-inverted serial signal.

> The voltage on the RX pin should not go below 0 V and should not exceed 5 V.

The Simple Motor Controller G2 provides logic-level (0 to 3.3 V) serial output on its serial transmit line, **TX**. The bytes sent by the motor controller on TX are typically responses to commands that request information, but they can also be data received by the TXIN pin and passed on. If you aren't interested in receiving TTL serial bytes from the motor controller, you can leave the TX line disconnected. See **Section 4.3** for more information on connecting a serial device to the Simple Motor Controller.

The serial interface is *asynchronous*, meaning that the sender and receiver are separately configured ahead of time to agree on the length of a bit (this is known as the "baud rate" and it is usually specified in bits per second, or bps), and each side independently times the serial bits. The Simple Motor Controller G2 has the ability to automatically detect the baud rate, which means that it can be used even when the baud rate of the serial source is unknown as long as the serial source initiates communication by sending the proper baud rate indication byte: 0xAA (written as 170 in decimal notation). The Simple Motor Controller G2 works with baud rates from 1200 to 500,000 bits per second. Asynchronous TTL serial is available as hardware modules called "UARTs" on many microcontrollers, but it can also be "bit-banged" by a standard digital output line under software control.

The data format is 8 data bits, no parity bit, and one stop bit, which is often expressed as **8-N-1**. The diagram below depicts a typical asynchronous, non-inverted TTL serial byte:

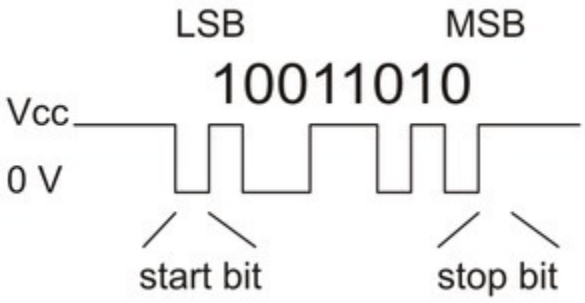


Diagram of a non-inverted TTL serial byte.

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 (3.3 V) 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. The Simple Motor Controller G2 supports fixed baud rates of 1099 bps to 2 Mbps and can automatically detect baud rates up to 500 kbps in auto-detect baud rate mode.

You must wait for at least 1 ms after the Simple Motor Controller powers up or is reset before you start sending data. Anything sent during this first millisecond is likely to be ignored or incorrectly received.

Whenever connecting devices, remember to wire the grounds together, and ensure that each device is properly powered. Unpowered devices with a TTL serial port can turn on or partially on, drawing power from the serial line, which means that extra care must be taken when turning power off and on to reset the devices.

Note: TTL serial is **not** the same as RS-232 serial. You must use an inverter and level shifter such as a MAX232 or a **Pololu 23201a Serial Adapter [https://www.pololu.com/product/126]** if you want to interface an RS-232 device with the Simple Motor Controller. Connecting an RS-232 device directly to the Simple Motor Controller can permanently damage it.

For more information about the serial pins, see **Section 4.2**.

I²C

The Simple Motor Controller G2's I²C interface consists of its SDA pin and its SCL pin. The SDA pin is the same physical pin as the RX pin, and the SCL pin is linked to the TX pin, so you cannot use serial and I²C at the same time. I²C is disabled by default, so you must enable it by checking the "Enable I²C" checkbox in the "Input settings" tab of the Simple Motor Control Center G2. For more information about the I²C pins, see **Section 4.2**. For more information about connecting an I²C device, see **Section 4.4**.

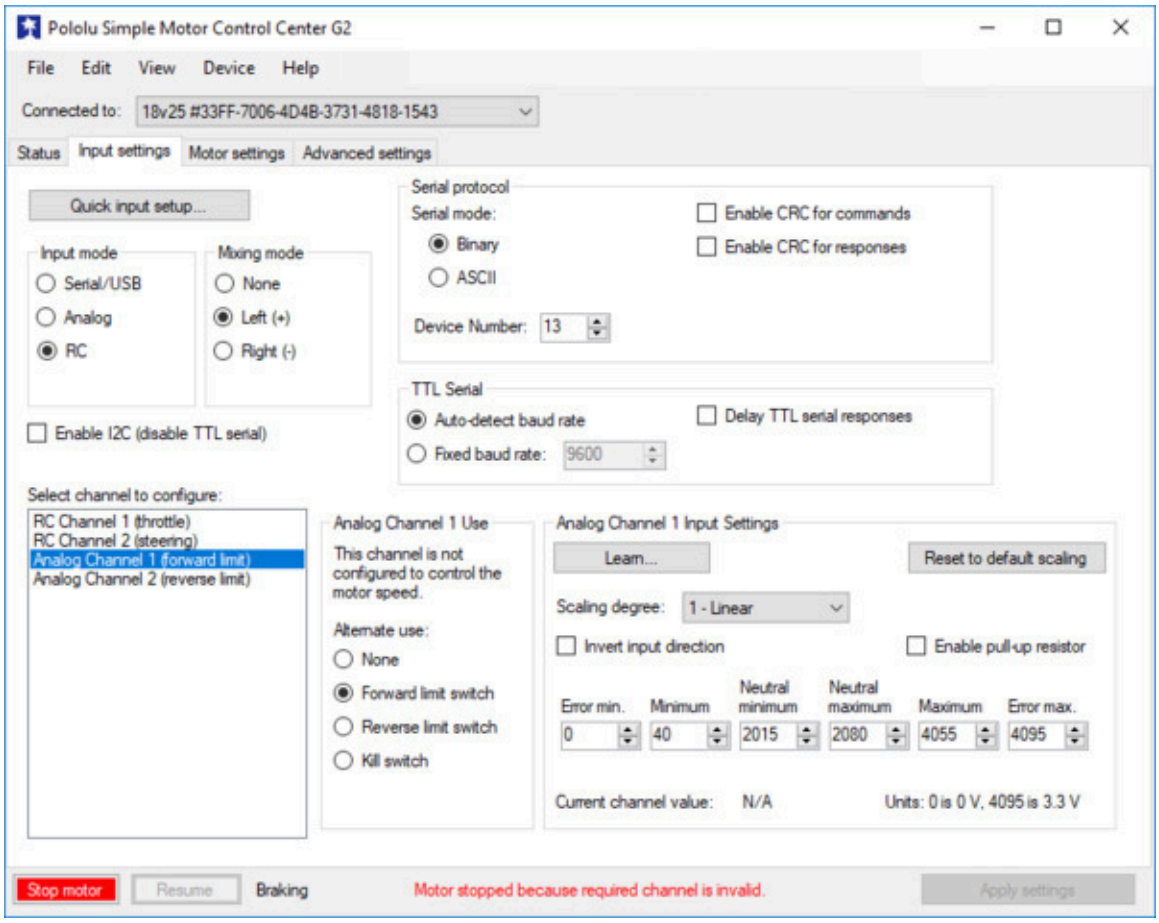
To send commands to the I²C interface, you would first send the I²C START condition, followed by the 7-bit device number of the controller you want to address, then a zero bit to indicate that you are going to write to the controller. Next, transmit all the bytes of one or more binary command packets. Then end the write transfer using a STOP condition. All of the bytes transferred also have acknowledgment bits, as described in the I²C specification.

If you want to read data from the controller using I²C, the command for reading the data should be the last command you send in the write transfer. After ending the write transfer, start a read transfer: send an I²C START condition, followed by the 7-bit device number, and then a one bit to indicate that you are reading. Then read as many bytes of the response as you want to read. Finally, end the read transfer using a STOP condition. It is OK to only read a portion of the response, or to read nothing.

The Simple Motor Controller G2 uses I²C clock stretching, meaning that it will drive the SCL pin low sometimes while it is busy handling a transfer.

6.1. Serial and I²C settings

The behavior of the Simple Motor Controller G2's USB virtual serial port, TTL serial port, and I²C interfaces is determined by a number of settings, almost all of which can be found under the "Input settings" tab of the Simple Motor Control Center G2:



Input settings tab in the Pololu Simple Motor Control Center G2.

The **Enable I2C (disable TTL serial)** option enables the I²C interface on the SDA and SCL pins, and disables the serial interface on the RX and TX pins. You should only enable this option if you intend to use I²C.

The **Serial mode** setting determines which protocol the Simple Motor Controller G2 will accept on its virtual serial port and TTL serial port. It does not affect I²C.

- **Binary:** In this mode, the controller expects command packets comprised of a series of bytes that conform to the Compact, Pololu, or Mini SSC protocol formats (see **Section 6.2** for more information on these protocols). The binary mode commands are more compact than their

ASCII mode counterparts, so they can be transmitted faster. This mode also lets you send commands addressed to a particular device number, so this mode should be used when multiple devices are daisy-chained together on the same serial line.

- **ASCII:** In this mode, the controller expects command packets comprised of ASCII characters, which makes the commands potentially more friendly to beginners since they look like character strings rather than seemingly random sets of bytes. Also, the ASCII protocol makes it easy to send commands to the Simple Motor Controller G2 from a terminal program. See **Section 6.3** for more information on the ASCII protocol.

When the **Enable CRC for commands** option is enabled, the Simple Motor Controller G2 requires a cyclic redundancy check (CRC) byte at the end of every binary mode command packet, which helps ensure that the controller won't misinterpret noisy commands or act up when presented with a stream of random serial bytes (see **Section 6.5** for more information on CRCs). If the CRC byte is not appended or is incorrect, the controller reports a serial CRC error. CRC is not available for ASCII mode commands.

When the **Enable CRC for responses** option is enabled, the Simple Motor Controller G2 will append a CRC byte onto the end of every binary mode response it generates, which lets you be more confident that the response was not corrupted by noise. CRC is not available for ASCII mode responses.

The **Device number** is a number between 0 and 127 that can be used to address this device in Pololu Protocol and Mini SSC protocol commands. This setting is useful when using the Simple Motor Controller G2 with other devices in a daisy-chained configuration (see **Section 6.6**). The "Device number" is also used as the 7-bit I²C address of the device, which must be transmitted as the first 7 bits of every I²C transfer. If you are connecting multiple controllers to the same serial bus or to the same I²C bus, each controller should have a different device number.

There are two options for determining the baud rate on the TTL serial port (the RX and TX pins). These settings do not apply to the USB virtual serial port or the I²C interface.

- **Auto-detect baud rate:** In this mode, the Simple Motor Controller G2 automatically detects the baud rate from the first **0xAA (170)** baud rate indication byte it receives on the RX line. Every time the controller is powered up or reset, and every time you apply new settings to the controller, you will need to send a baud rate indication byte before the controller will accept TTL serial commands. Once you send the baud rate indication byte, you can check the Status tab of the Simple Motor Control Center G2 to see what baud rate the controller detected. The controller can automatically detect baud rates from 1200 bps to 500 kbps. This mode is only available when the serial mode is "Binary"; the fixed baud rate option is automatically selected when the serial mode is "ASCII".
- **Fixed baud rate:** In this mode, the Simple Motor Controller G2 will only respond to TTL serial

signals transmitted at the configured fixed baud rate (in units of bits per second, or bps). The fixed baud rate can be set from 1099 bps to 2 Mbps, but the Simple Motor Controller will not be able to keep up with a constant stream of commands at baud rates over 500 kbps (if you send commands to the controller faster than it can process them, the receive buffer will eventually fill up, data will be lost, and a serial RX overrun error will be generated).

Enabling the **Delay TTL serial responses** feature causes the Simple Motor Controller G2 to wait for approximately 4 ms before transmitting a TTL serial response. This is useful when interfacing with devices like the Basic Stamp that use half-duplex UARTs and need time to switch from transmit mode to receive mode. When this feature is disabled, transmission of a response packet begins as soon as possible after the last byte of a command packet is received (if that command packet generates a response).

The **Command timeout** setting lets you configure the Simple Motor Controller to shut down the motor if too much time elapses between received commands, which could happen if your serial control source gets disconnected or loses power. It is located under the "Advanced settings" tab of the Simple Motor Control Center G2. See **Section 5.3** for more information on this parameter.

6.2. Binary commands

This section describes the binary mode commands supported by the Simple Motor Controller G2. These commands are used on the TTL serial port and USB virtual serial port when the "Serial mode" setting is set to "Binary". The I²C interface only accepts the commands documented here.

Communication is achieved by sending command packets consisting of a single command byte followed by any data bytes that command requires (not all commands require data bytes; some command packets simply consist of a single command byte). Command bytes always have their most significant bits set, while data bytes almost always have their most significant bits cleared:

0x80 (128) ≤ *command byte* ≤ 0xFF (255)
0x00 (0) ≤ *data byte* ≤ 0x7F (127)

This means that each data byte can only transmit seven bits of information. The only exception to this is the Mini SSC command, where the command byte is 0xFF, or 255, and the data bytes can have any value from 0x00 to 0xFE (0 to 254).

Some commands are used to read data from the controller, so the controller will generate a response packet. The bytes in the response packet can have any value.

Note: If you are using the TTL serial interface and the motor controller is in auto-detect baud rate mode, you must send the baud rate indication byte **0xAA**, or 170, on the RX line before sending any commands. The 0xAA baud rate indication byte can be the first byte of a Pololu protocol command, or it can be transmitted as a single byte. Communication via the controller's USB virtual serial port or I²C is unaffected by baud rate settings and does not require the transmission of an initial baud rate indication byte.

This guide displays byte values in the format: **"hex (decimal)"**, where **hex** is the **hexadecimal** [http://simple.wikipedia.org/wiki/Hexadecimal] (base-16) representation of the byte's value, and **decimal** is the decimal representation of the byte's value. The hexadecimal representation starts with the prefix **0x** (e.g. 0x10).

Keep in mind that a byte is simply a number between 0x00 (0) and 0xFF (255). For these protocols, the important thing about a byte is its value, not the notation (e.g. hex, decimal, or binary) you use in your source code to write the byte.

The following three sub-protocols are available in binary serial mode:

Compact Protocol

In general, this is the protocol that you should use if a single Simple Motor Controller G2 is the only device that will receive the bytes in your command packets (which is always the case if you are using the I²C or virtual USB serial port). The compact protocol command packet is simply:

command byte (MSB set)	[data byte 1]	[data byte 2]	...	[data byte n]
1XXXXXXX	[0XXXXXXX]	[0XXXXXXX]	...	[0XXXXXXX]

For example, if we want to set the motor speed to 3200 (full speed) forward, we could send the following byte sequence:

Hex notation:	**0x85,**	**0x00,**	**0x64**
Decimal notation:	**133,**	**0,**	**100**

The byte 0x85 is the Set Motor Forward command, and the last two bytes contain the speed.

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 Simple Motor Controller on a single serial line along with our

other serial controllers (including additional Simple Motor Controllers) and, using this protocol, send commands specifically to the desired controller without confusing the other devices on the line.

To use the Pololu protocol, you must transmit 0xAA (170 in decimal) as the first (command) byte, followed by a device number data byte. The default device number for the Simple Motor Controller is **0x0D** (**13** in decimal), but this is a setting you can change. Any controller on the line whose device number matches the specified device number accepts the command that follows; all other Pololu devices 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 (170)	device number	command byte (with MSB cleared)	[data byte 1]	[data byte 2]	...	[data byte n]
10101010	0XXXXXXX	0XXXXXXX	[0XXXXXXX]	[0XXXXXXX]	...	[0XXXXXXX]

For example, if we want to set the motor speed to 3200 (full speed) forward, we could send the following byte sequence:

Hex notation:	**0xAA,**	**0x0D,**	**0x05,**	**0x00,**	**0x64**
Decimal notation:	**170,**	**13,**	**5,**	**0,**	**100**

The byte 0x05 is the Set Motor Forward command (0x85) with its most significant bit cleared.

Mini SSC protocol

The Simple Motor Controller G2 also responds to the Scott Edwards Mini SSC protocol, a simple, three-byte serial protocol commonly used by servo controllers. Since it only takes three serial bytes to set the speed of one motor, this protocol is good if you need to send many commands rapidly to multiple motor controllers. The Mini SSC protocol is to transmit 0xFF (255 in decimal) as the first (command) byte, followed by a device number byte and an 8-bit motor speed byte. If you think of the available motor speeds as ranging from -127 to +127, the motor speed byte is this signed speed value offset by 127. Therefore, a speed byte of 0 results in full-speed reverse, a speed byte of 127 results in speed 0 (motor stopped), and a speed byte of 254 results in full-speed forward. The command packet is:

0xFF (255)	device number (0-254)	speed byte (0-254)
11111111	XXXXXXXX	XXXXXXXX

For example, if we want to set the speed of device 13 to approximately half-speed forward (63+127=190), we could send the following byte sequence:

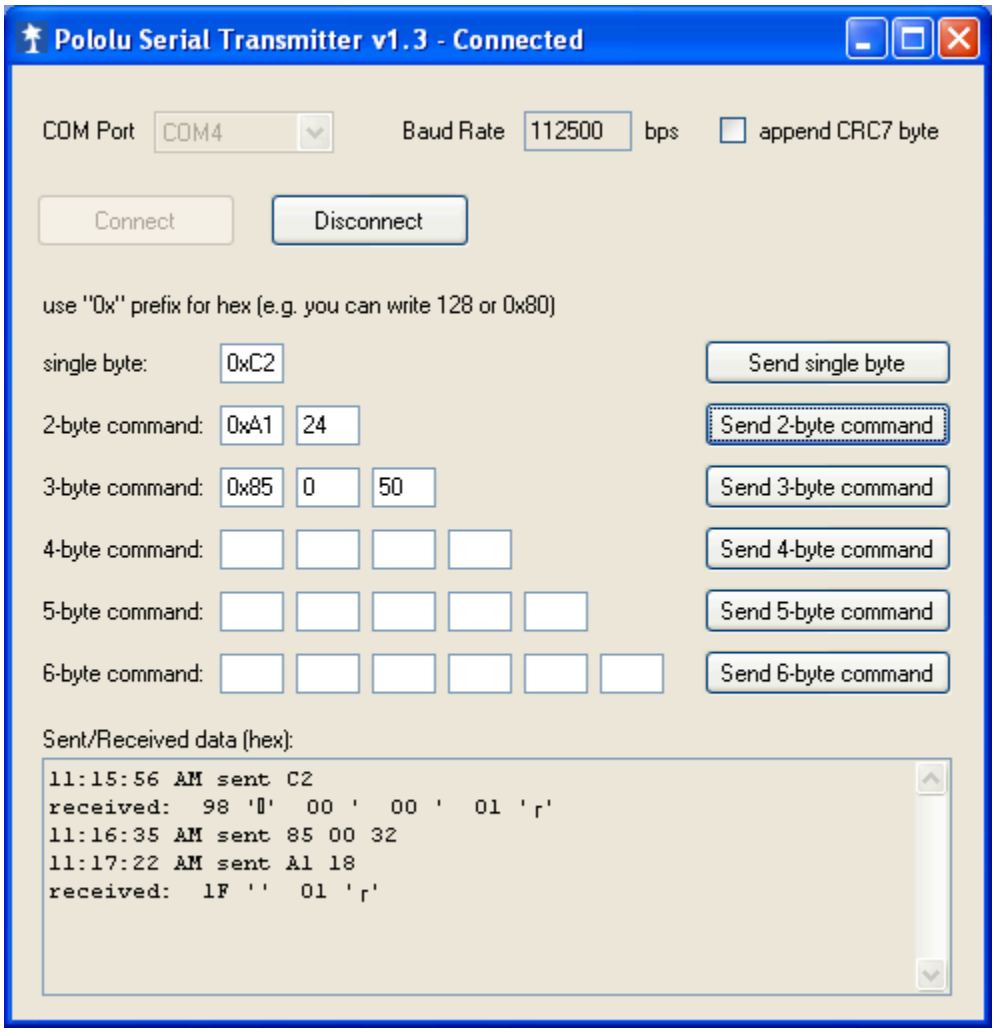
Hex notation:	**0xFF,**	**0x11,**	**0xBE**
Decimal notation:	**255,**	**13,**	**190**

The device number byte and speed byte can be any value except 255, though the Simple Motor Control Center G2 will not let you set the controller's device number to a value greater than 127. If the device number byte matches the motor controller's device number or if the device number byte is 254, the motor controller will respond to the command (all controllers respond to Mini SSC commands addressed to Device Number 254).

The Simple Motor Controller G2 identifies the Pololu, Compact, and Mini-SSC protocols on the fly; you can freely mix commands in the three protocols.

Trying the Binary Serial Interface

If you are having trouble using the Binary protocols, it can help to first use a program like the **Pololu Serial Transmitter utility for Windows** **[https://www.pololu.com/docs/0J23]** to send bytes to the Simple Motor Controller G2's virtual COM port. This program makes it easy send packets of arbitrary bytes, which can help you identify if your problems are with your control software or with the bytes you are trying to send. The Serial Transmitter Utility can even automatically append the appropriate CRC7 byte to the end of the transmitted command packet.



Sending Binary (Compact Protocol) commands to the Simple Motor Controller with the Pololu Serial Transmitter utility.

6.2.1. Binary command reference

Exit safe-start (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2
Compact Protocol	**0x83 (131)**	-	-
Pololu Protocol	0xAA (170)	device number	**0x03 (3)**

Description: If the input mode is Serial/USB, and you have not disabled safe-start protection, then

this command is required before the motor can run. Specifically, this command must be issued when the controller is first powered up, after any reset, and after any error stops the motor. This command has no serial response.

If you just want your motor to run whenever possible, you can transmit exit safe start and motor speed commands regularly. The motor speed commands are documented below. One potential problem with this approach is that if there is an error (e.g. the battery becomes disconnected) then the motor will start running immediately when the error has been resolved (e.g. the battery is reconnected).

If you want to prevent your motor from starting up unexpectedly after the controller has recovered from an error, then you should only send an exit safe start command after either waiting for user input or issuing a warning to the user.

Motor forward (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4
Compact Protocol	**0x85 (133)**	speed byte 1	speed byte 2	-	-
Compact alternate use	**0x85 (133)**	0	speed %	-	-
Pololu Protocol	0xAA (170)	device number	**0x05 (5)**	speed byte 1	speed byte 2
Pololu alternate use	0xAA (170)	device number	**0x05 (5)**	0	speed %

Description: This command lets you set the full-resolution motor target speed in the forward direction. The motor speed must be a number from 0 (motor stopped) to 3200 (motor forward at full speed) and is specified using two data bytes. The first data byte contains the **low five bits** of the speed and the second data byte contains the **high seven bits** of the speed.

The first speed data byte can be computed by taking the full (0 to 3200) speed **modulo [http://simple.wikipedia.org/wiki/Modular_arithmetic]** 32, which is the same as dividing the speed by 32, discarding the quotient, and keeping only the remainder. We can get the same result using binary math by bitwise-ANDing the speed with 0x1F (31). In C (and many other programming languages), these operations can be carried out with the following expressions:

```
speed_byte_1 = speed % 32;
```

or, equivalently:

```
speed_byte_1 = speed & 0x1F;
```

The second speed data byte can be computed by dividing the full (0 to 3200) speed by 32, discarding the remainder, and keeping only the quotient (i.e. turn the division result into a whole number by dropping everything after the decimal point). We can get the same result using binary math by bit-shifting the speed right five places. In C (and many other programming languages), these operations can be carried out with the following expressions:

```
speed_byte_2 = speed / 32;
```

or, equivalently:

```
speed_byte_2 = speed >> 5;
```

This command has no serial response.

Example:

If we want to set the motor target speed to half-speed forward, we can use the above equations to compute that the first speed byte must be the remainder of 1600/32, or **0**, and the second speed byte must be the quotient of 1600/32, or **50**. Therefore, we can send the following compact protocol bytes:

We send:

	Command Byte	Data Byte 1	Data Byte 2
Compact Protocol	0x85 (133)	0x00 (0)	0x32 (50)

Alternate interpretation: The allowed values for the second speed data byte are 0–100, so you can ignore the first speed data byte (always set it to 0), and consider the second data byte to simply be the speed percentage. For example, to drive the motor at 53% speed, you would use byte1=0 and byte2=53.

Motor reverse (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4
Compact Protocol	**0x86 (134)**	speed byte 1	speed byte 2	-	-
Compact Alternate Use	**0x86 (134)**	0	speed %	-	-
Pololu Protocol	0xAA (170)	device number	**0x06 (6)**	speed byte 1	speed byte 2
Pololu Alternate Use	0xAA (170)	device number	**0x06 (6)**	0	speed %

Description: This command lets you set the full-resolution motor target speed in the reverse direction. The motor speed must be a number from 0 (motor stopped) to 3200 (motor reverse at full speed) and is specified using two data bytes, the first containing the low five bits of the speed and the second containing the high seven bits of the speed. This command behaves the same as the Motor Forward command except the motor moves in the opposite direction.

Motor forward 7-Bit (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3
Compact Protocol	**0x89 (137)**	speed	-	-
Pololu Protocol	0xAA (170)	device number	**0x09 (9)**	speed

Description: This command sets the motor target speed in the forward direction based on the specified low-resolution (7-bit) Speed byte. The Speed byte is a number from 0 (motor stopped) to 127 (motor forward at full speed). This command has no serial response.

Example:

To set the motor target speed to approximately half-speed forward (63), we could send the following compact protocol bytes:

We send:

	Command Byte	Data Byte 1
Compact Protocol	0x89 (137)	0x3F (63)

Motor Reverse 7-Bit (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3
Compact Protocol	**0x8A (138)**	speed	-	-
Pololu Protocol	0xAA (170)	device number	**0x0A (10)**	speed

Description: This command sets the motor target speed in the reverse direction based on the specified low-resolution (7-bit) speed byte. The speed byte is a number from 0 (motor stopped) to 127 (motor reverse at full speed). This command has no serial response.

Set speed Mini SSC (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2
Mini SSC Protocol	**0xFF (255)**	device number	speed

Description: This is the Mini SSC Protocol command for setting the motor speed. This command has no serial response. See **Section 6.2** for complete documentation of this command.

Motor Brake (Serial/USB input mode only)

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3
Compact Protocol	**0x92 (146)**	brake amount	-	-
Pololu Protocol	0xAA (170)	device number	**0x12 (18)**	brake amount

Description: This command causes the motor to immediately brake or coast (configured deceleration limits are ignored). The brake amount byte can have a value from 0 to 32, with 0 resulting in coasting (the motor outputs are disabled) and any non-zero value resulting in braking (the motor outputs are driven low). Requesting a brake amount greater than 32 results in a serial format error. This command has no serial response.

Example:

To make the motor brake, we would transmit the following compact protocol bytes:

We send:

	Command Byte	Data Byte 1
Compact Protocol	0x92 (146)	0x20 (32)

Get variable

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3
Compact Protocol	**0xA1 (161)**	variable ID	-	-
Pololu Protocol	0xAA (170)	device number	**0x21 (33)**	variable ID

Response format:

Response Byte 1	Response Byte 2
variable low byte	variable high byte

Description: This command lets you read a 16-bit variable from the Simple Motor Controller. See **Section 6.4** for a list of all of available variables. The value of the requested variable is transmitted as two bytes, with the low byte sent first. You can reconstruct the variable value from these bytes using the following equation:

```
variable_low_byte + 256 * variable_high_byte
```

If the variable type is *signed* and the above result is greater than 32767, you will need to subtract 65536 from the result to obtain the correct, signed value. Alternatively, if it is supported by the language you are using, you can cast the result to a signed 16-bit data type.

Requesting an invalid variable ID results in a serial format error, and the controller does not transmit a response.

Example:

To request the temperature A reading (variable ID 24), we would transmit the following compact protocol bytes and wait until we have received two bytes in response from the Simple Motor

Controller G2:

We send:

	Command Byte	Data Byte 1
Compact Protocol	0xA1 (161)	0x18 (24)

We receive:

Response Byte 1	Response Byte 2
0x1E (30)	0x01 (1)

This response tells us that the temperature is:

```
30 + 256 * 1 = 286
```

in units of 0.1 °C, which means the temperature is **28.6 °C**.

Set motor limit

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4	Data Byte 5
Compact Protocol	**0xA2 (162)**	limit ID	limit byte 1	limit byte 2	-	-
Pololu Protocol	0xAA (170)	device number	**0x22 (34)**	limit ID	limit byte 1	limit byte 2

Response format:

Response Byte 1
response code

Description: This command lets you change the temporary motor limit variables documented in **Section 6.4**. The ID of the limit to set is specified by the first compact protocol data byte, and the value of the limit is specified by the next two data bytes, the first of which (limit byte 1) contains the **low seven bits** of the value and the second (limit byte 2) contains the **high seven bits**. Limit IDs from 0 to 3 affect both forward and reverse limits equally (they are "symmetric"). Limit IDs from 4 to

7 affect only forward limits and limit IDs from 8 to 11 affect only reverse limits. The following table provides the limit IDs for all of the temporary motor limit variables along with the allowed limit values:

ID	Name	Allowed values	Units
0 or 4	Max speed forward	0–3200	0=0%, 3200=100%
1 or 5	Max acceleration forward	0–3200 (0=no limit)	Δ*speed* per *update period*
2 or 6	Max deceleration forward	0–3200 (0=no limit)	Δ*speed* per *update period*
3 or 7	Brake duration forward	0–16384	4 ms
0 or 8	Max speed reverse	0–3200	0=0%, 3200=100%
1 or 9	Max acceleration reverse	0–3200 (0=no limit)	Δ*speed* per *update period*
2 or 10	Max deceleration reverse	0–3200 (0=no limit)	Δ*speed* per *update period*
3 or 11	Brake duration reverse	0–16384	4 ms

Note: The brake duration units used by this command are **4 ms**, which differs from 1 ms units used by the brake duration *variables* returned by the "Get variable" command.

The first limit value byte, **limit byte 1**, can be computed by taking the full limit value modulo (or "mod") 128, which is the same as dividing the value by 128, discarding the quotient, and keeping only the remainder. We can get the same result using binary math by bitwise-ANDing the limit with 0x7F (127). In C (and many other programming languages), these operations can be carried out with the following expressions:

```
limit_byte_1 = limit % 128;
```

or, equivalently:

```
limit_byte_1 = limit & 0x7F;
```

The second limit value byte, **limit byte 2**, can be computed by dividing the full limit value by 128, discarding the remainder, and keeping only the quotient (i.e. turn the division result into a whole number by dropping everything after the decimal point). We can get the same result using binary

math by bit-shifting the limit right seven places. In C (and many other programming languages), these operations can be carried out with the following expressions:

```
limit_byte_2 = limit / 128;
```

or, equivalently:

```
limit_byte_2 = limit >> 7;
```

Note that the hard motor limit settings place restrictions on the limit values you can set with this command (see **Section 5.2** for more information on the hard motor limits). The hard limits configured through the Simple Motor Control Center G2 are considered minimal safety requirements, and the temporary limits cannot be changed in a way that makes the controller "less safe" than this. This means that the maximum speed, acceleration, and deceleration temporary limits cannot be increased beyond their hard-limit counterparts and the brake duration limits cannot be decreased below their hard-limit counterparts. If you try to set a temporary limit in a way prohibited by the corresponding hard limit, the temporary limit value is set to the hard limit and the response code byte indicates that the value could not be set as requested.

If the arguments to this command are valid, the controller responds to this command with a single-byte code:

Response code	Description
0	No problems setting the limit.
1	Unable to set forward limit to the specified value because of hard motor limit settings.
2	Unable to set reverse limit to the specified value because of hard motor limit settings.
3	Unable to set forward and reverse limits to the specified value because of hard motor limit settings.

Limit IDs above 11 and limit values outside of their allowed value ranges result in a serial format error and no response is transmitted by the controller.

The limit values set with this command persist only until the controller is next reset or the "Apply settings" button is next clicked in the Simple Motor Control Center G2, at which point the temporary limit settings are all reinitialized to the hard limit settings.

Example:

To set the reverse deceleration limit (limit ID 10) to 500, we can use the above equations to compute that **limit byte 1** must be the remainder of 500/128, or **116**, and **limit byte 2** must be the quotient of 500/128, or **3**. Therefore, we can send the following compact protocol bytes and wait until we have received one byte in response from the Simple Motor Controller G2:

We send:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3
Compact Protocol	0xA2 (162)	0x0A (10)	0x74 (116)	0x03 (3)

We receive:

Response Byte 1
0x00 (0)

This response tells us the temporary limit was set as requested. If our max deceleration reverse hard motor limit was below 500, we would receive a response code of **2**, which would tell us that the temporary limit was not set as requested (rather, it was set equal to whatever the hard limit is).

Set current limit

Command format:

	Command Byte	Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4
Compact Protocol	**0x91 (145)**	current byte 1	current byte 2	-	-
Pololu Protocol	0xAA (170)	device number	**0x11 (17)**	current byte 1	current byte 2

Description: This command lets you change the hardware current limit threshold temporarily. See **Section 5.2** for information about hardware current limiting and how to calculate a value to use with this command.

The value of the current limit is specified by the two data bytes. The first (current byte 1) contains the **low seven bits** of the value, and the second (current byte 2) contains the **high seven bits**.

The first data byte, **current byte 1**, can be computed by taking the full value modulo (or "mod")

128, which is the same as dividing the value by 128, discarding the quotient, and keeping only the remainder. We can get the same result using binary math by bitwise-ANDing the limit with 0x7F (127). In C (and many other programming languages), these operations can be carried out with the following expressions:

```
limit_byte_1 = limit % 128;
```

or, equivalently:

```
limit_byte_1 = limit & 0x7F;
```

The second limit value byte, **limit byte 2**, can be computed by dividing the value by 128, discarding the remainder, and keeping only the quotient (i.e. turn the division result into a whole number by dropping everything after the decimal point). We can get the same result using binary math by bit-shifting the limit right seven places. In C (and many other programming languages), these operations can be carried out with the following expressions:

```
limit_byte_2 = limit / 128;
```

or, equivalently:

```
limit_byte_2 = limit >> 7;
```

The current limit set with this command persists only until the controller is next reset or the "Apply settings" button is next clicked in the Simple Motor Control Center G2, at which point the current limit is reinitialized to the value stored in the settings.

Get firmware version

Command format:

	Command Byte	Data Byte 1	Data Byte 2
Compact Protocol	**0xC2 (194)**	-	-
Pololu Protocol	0xAA (170)	device number	**0x42 (66)**

Response format:

Response Byte 1	Response Byte 2	Response Byte 3	Response Byte 4
product ID low byte	product ID high byte	minor FW version (BCD format)	major FW version (BCD format)

Description: This command lets you read the Simple Motor Controller G2 product number and firmware version number. The first two bytes of the response are the low and high bytes of the product ID (each Simple Motor Controller G2 version has a unique product ID), and the last two bytes of the response are the firmware minor and major version numbers in **binary-coded decimal (BCD) format** **[http://en.wikipedia.org/wiki/Binary-coded_decimal]**. BCD format means that the version number is the value you get when you write it in hex and then read it as if it were in decimal. For example, a minor version byte of 0x15 (21) means a the minor version number is 15, not 21.

Example:

To request the product ID and firmware version, we would transmit the following compact protocol byte and wait until we have received four bytes in response from the Simple Motor Controller (or until our receiving function times out, which could happen if there is a problem):

We send:

	Command Byte
Compact Protocol	0xC2 (194)

We receive:

Response Byte 1	Response Byte 2	Response Byte 1	Response Byte 2
0x98 (152)	0x00 (0)	0x00 (0)	0x01 (1)

This response tells us that the product ID is **0x00A3 (152)** and the firmware version is **1.0**.

Stop motor

Command format:

	Command Byte	Data Byte 1	Data Byte 2
Compact Protocol	**0xE0 (224)**	-	-
Pololu Protocol	0xAA (170)	device number	**0x60 (96)**

Description: This command sets the motor target speed to zero and makes the controller susceptible to a safe-start violation error if safe start is enabled. Put another way, this command will stop the motor (configured deceleration limits will be respected) and not allow the motor to start again until the Safe-Start conditions required by the Input Mode are satisfied. This command has no serial response.

6.3. ASCII commands

When configured in "ASCII" serial mode, the Simple Motor Controller G2 TTL serial port and USB virtual serial port offer a simple serial interface based on ASCII characters. This mode makes it easy to interact with the Simple Motor Controller G2 through a terminal program, such as Tera Term, and it can provide a more intuitive interface for users who would rather deal with character strings than bits and bytes.

There are some limitations when using ASCII mode, however:

- The commands are longer than their binary mode Compact Protocol counterparts, so they will take longer to send.
- Automatic baud detection is not available; you must configure the Simple Motor Controller G2 to the appropriate fixed baud rate ahead of time if you are communicating using TTL serial.
- CRC error detection is not available.
- The ASCII mode serial responses might be harder to parse with some programming languages than the binary mode responses.

Command format

ASCII commands consist of a command string, which is typically a single letter, followed by a comma-separated list of numbers representing the arguments to the command. Not all commands take arguments, and only one command (Set motor limit) takes multiple arguments. All commands must be terminated by a special termination character, such as a carriage return (<CR>).

Expressed generally, the format is:

```
command string + [argument 1 + [',' + argument 2]] + termination character
```

For example, to command the motor to drive forward at speed 3200 (full speed), we could send the following ASCII command:

"F3200<CR>"

Here the command string is **"F"**, the argument string is **"3200"**, and the termination character is **<CR>**.

ASCII commands are case-insensitive and white-space is ignored, so **"F3200<CR>"** has the same effect as **"f 3200 <CR>"**.

The specific commands are documented in **Section 6.3.1**.

Command strings

The following table lists all of the available command strings:

Command string	Command name
"GO"	Exit safe-start
"F"	Motor Forward
"R"	Motor Reverse
"B"	Motor Brake
"D"	Get Variable
"L"	Set Motor Limit
"V"	Get Firmware Version
"X"	Stop Motor

Argument strings

Command arguments are expressed as strings of ASCII digits. By default, the string is interpreted as a decimal (base 10) value, but an "H" can be appended to the end of the string to tell the Simple Motor Controller G2 to interpret it as a hexadecimal (base 16, or hex) value. For example, you can represent an argument value of 127 with **"127"** or **"7FH"** (0x7F is the hex representation of 127). The arguments to the three motor commands ("F", "R", and "B") can also be written as percentages by appending a "%" to the end of the argument. For example, you can represent full motor speed with the argument **"3200"** or with the argument **"100%"**.

Termination Characters

ASCII mode accepts three different termination characters:

- **Carriage return:** A carriage return is the character typically sent when you press the Enter key in a terminal program. It is often written as **<CR>** and has a character value of 13. In C, this special character can be written as **'\r'**.

- **NL line feed:** Also known as **"new line"**, this character is often written as **<LF>** and has a character value of 12. In C, this special character can be written as **'\n'**.
- **Null character:** This character is used to terminate strings in C. It is often written as **<NUL>** and has a character value of 0. In C, the string "abc" is comprised of the four characters: 'a', 'b', 'c', and <NUL>.

One of the above three characters must be the last character in your ASCII command string.

Responses

Any ASCII mode command string that contains more than just a termination character will generate a serial response from the Simple Motor Controller. The first character of the response gives you information about the status of the controller; it can be one three possible characters:

Status character	Meaning
'.'	The last command was understood and no errors are stopping the motor.
'!'	The last command was understood and errors are stopping the motor.
'?'	The last command was not understood (a serial format error has occurred).

If the command sent responds with data (e.g. the "Get variable" command), the data follows the status character as a decimal (base 10) string of ASCII digits.

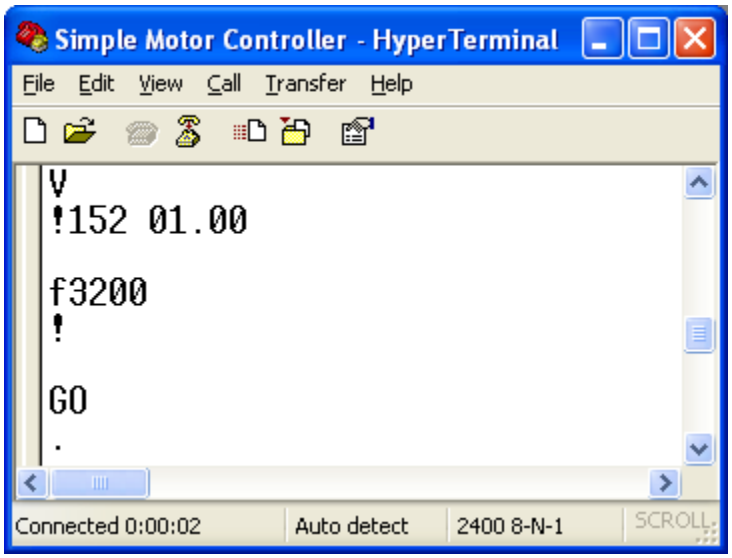
The ASCII mode serial response is always terminated by a carriage return (<CR>) followed by a line feed (<LF>).

For example, if we send a "Motor forward" command while no errors are stopping the motor, the response would be **".<CR><LF>"**. If we send a "Get variable" command while errors are stopping the motor, the response might be **"!123<CR><LF>"**, which would indicate that the requested variable has a value of 123.

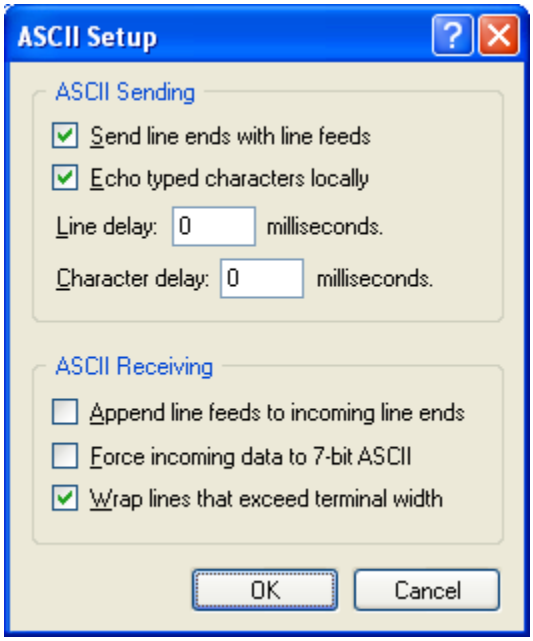
Commands that consist only of termination characters do not result in a serial response from the Simple Motor Controller G2. All other commands, even invalid ones, cause the Simple Motor Controller G2 to respond when a termination character is received.

Using a terminal program

ASCII mode makes it easy to communicate with the Simple Motor Controller from a terminal program, such as Tera Term or HyperTerminal. The responses are formatted so that they will appear nicely in the terminal window.



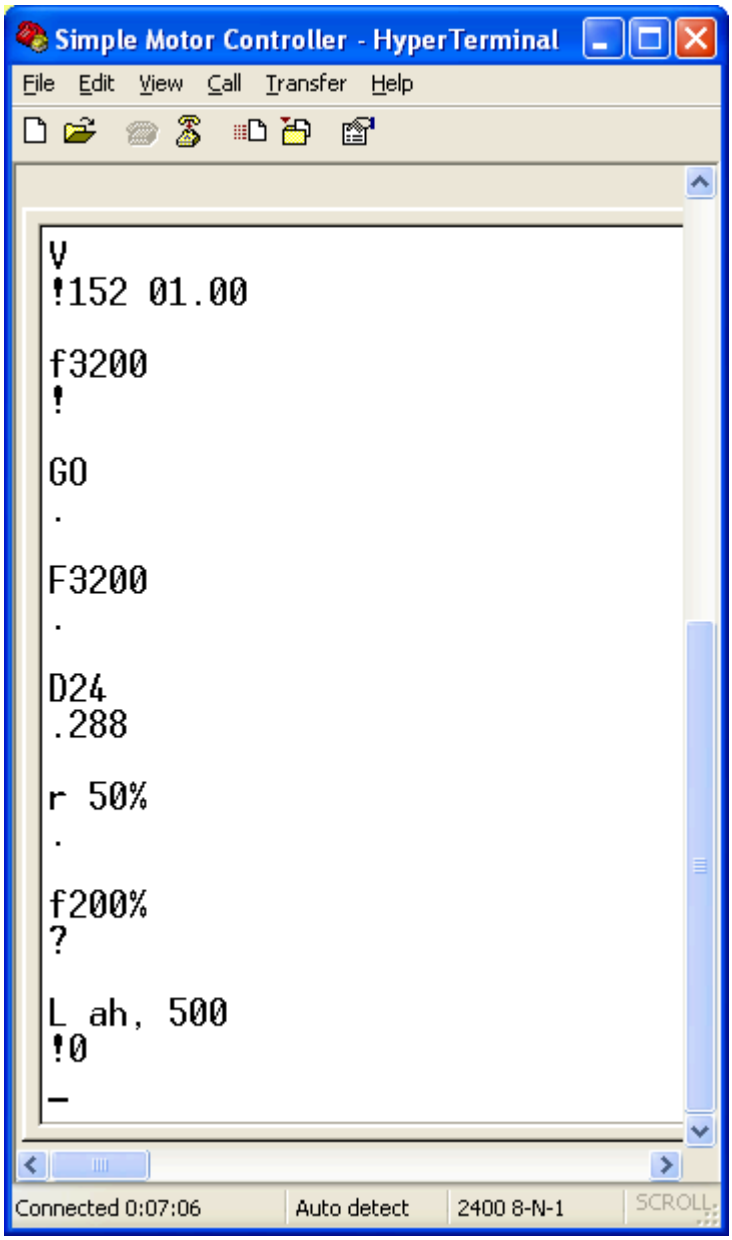
We recommend you enable local echoing of transmitted characters when typing commands into a terminal program. The following picture shows our recommended ASCII settings when using HyperTerminal:



You can get to this dialog by going to the **File > Properties** menu and clicking on the **ASCII Setup…** button under the Settings tab.

6.3.1. ASCII command reference

Exit safe-start (Serial/USB input mode only)



Sending ASCII commands to the Simple Motor Controller from HyperTerminal (with echoing of typed characters enabled).

Command format: "GO<CR>"

Description: This command clears the Serial/USB safe-start violation and allows the motor to run. When safe-start protection is enabled, this command must be issued when the controller is first powered up, after any reset, and after any error stops the motor.

Motor forward (Serial/USB input mode only)

Command format: "F< `speed` ><CR>"

Description: This command sets the motor target speed in the forward direction. The argument `speed` can be an integer from 0 (motor stopped) to 3200 (motor forward at full speed) or an integer percentage from 0% to 100%. You can represent the speed in hex by putting an "H" at the end of the number. If the argument `speed` is outside the allowed range, a Serial Format Error occurs..

Examples: The following commands all make the motor drive forward at half speed:

- "F1600<CR>"
- "F50%<CR>"
- "F640H<CR>"

Motor reverse (Serial/USB input mode only)

Command format: "R< `speed` ><CR>"

Description: This command sets the motor target speed in the reverse direction. It behaves the same as the Motor Forward command above, except the motor turns in the opposite direction.

Motor brake (Serial/USB input mode only)

Command format: "B< `brake_amount` ><CR>"

Description: This command causes the motor to immediately brake by the specified amount (configured deceleration limits are ignored). The argument `brake_amount` can be an integer from 0 (maximum coasting) to 32 (full braking) or an integer percentage from 0% to 100%. You can represent the brake amount in hex by putting an "H" at the end of the number. If the argument `brake_amount` is outside the allowed range, a Serial Format Error occurs.

Error. **Examples:** The following commands all make the motor brake as hard as possible:

- "B32<CR>"
- "B100%<CR>"
- "B20H<CR>"

Get variable

Command Format: "D< `variable_id` ><CR>"

Description: This command lets you read a variable from the Simple Motor Controller G2. See **Section 6.4** for a list of the available variables. The value of the requested variable is transmitted as an ASCII-encoded decimal number. If `variable_id` argument is invalid, a serial format error occurs.

Example: The following commands both request the board temperature (variable ID 24, or 0x18):

- "D24<CR>"
- "D18H<CR>"

We might receive **".286<CR><LF>"** as a response. The leading '.' is a status character that indicates the last command was understood and no errors are currently stopping the motors. The rest of the characters before the carriage return (<CR>) and new line (<LF>) characters are an ASCII representation of a decimal (base 10) number. This particular variable has units of 0.1 °C, which would mean that the board temperature is **28.6 °C**.

Set motor limit

Command format: "L< `limit_id` >,< `limit_value` ><CR>"

Description: This command lets you change the temporary motor limit variables documented in **Section 6.4**. Limit IDs from 0 to 3 are affect both forward and reverse limits equally (they are "symmetric"). Limit IDs from 4 to 7 affect only forward limits and limit IDs from 8 to 11 affect only reverse limits. The following table provides the limit IDs for all of the temporary motor limit variables along with the allowed limit values:

ID	Name	Allowed values	Units
0 or 4	Max speed forward	0–3200	0=0%, 3200=100%
1 or 5	Max acceleration forward	0–3200 (0=no limit)	Δ*speed* per *update period*
2 or 6	Max deceleration forward	0–3200 (0=no limit)	Δ*speed* per *update period*
3 or 7	Brake duration forward	0–16384	4 ms
0 or 8	Max speed reverse	0–3200	0=0%, 3200=100%
1 or 9	Max acceleration reverse	0–3200 (0=no limit)	Δ*speed* per *update period*
2 or 10	Max deceleration reverse	0–3200 (0=no limit)	Δ*speed* per *update period*
3 or 11	Brake duration reverse	0–16384	4 ms

Note: The brake duration units used by this command are **4 ms**, which differs from 1 ms units used by the brake duration *variables* returned by the "Get variable" command.

Note that the hard motor limit settings place restrictions on the limit values you can set with this command (see **Section 5.2** for more information on the hard motor limits). The hard limits configured through the Simple Motor Control Center G2 are considered minimal safety requirements, and the temporary limits cannot be changed in a way that makes the controller "less safe" than this. This means that the maximum speed, acceleration, and deceleration temporary limits cannot be increased beyond their hard-limit counterparts and the Brake Duration limits cannot be decreased below their hard-limit counterparts. If you try to set a temporary limit in a way prohibited by the corresponding hard limit, the temporary limit value is set to the hard limit and the response code byte indicates that the value could not be set as requested.

If the arguments to this command are valid, the controller responds to this command with an ASCII digit:

Response Code	Description
'0'	No problems setting the limit.
'1'	Unable to set forward limit to the specified value because of hard motor limit settings.
'2'	Unable to set reverse limit to the specified value because of hard motor limit settings.
'3'	Unable to set forward and reverse limits to the specified value because of hard motor limit settings.

Limit IDs above 11 and limit values outside of their allowed value ranges result in a serial format error.

The limit values set with this command persist only until the controller is next reset or the "Apply settings" button is next clicked in the Simple Motor Control Center G2, at which point the temporary limit settings are all reinitialized to the hard limit settings.

Example: The following commands all set the reverse deceleration limit (limit ID 10, or 0x0A) to 500, or 0x1F4:

- "L10,500<CR>"
- "LAH,1F4H<CR>"
- "L10,1F4H<CR>"

The controller might send **".0<CR><LF>"** as a response. The leading '.' is a status character that indicates the last command was understood and no errors are currently stopping the motors. The following character, '0', means that the temporary limit was set as requested. If our max deceleration reverse hard motor limit was below 500, this character would have been '2', which would tell us that the temporary limit was not set as requested (rather, it was set equal to whatever the hard limit is).

Get firmware version

Command format: "V<CR>"

Description: This command prints the Simple Motor Controller G2 product number (in decimal) and firmware version number (the two major firmware version digits followed by the two minor firmware version digits). For example, the response to this command might be **".163 01.00<CR><LF>"**, which indicates a product ID of 163, a major firmware version of 1, and a minor firmware version of 0.

Stop motor

Command format: "X<CR>"

Description: This command sets the motor target speed to zero and makes the controller susceptible to a safe-start violation error if safe start is enabled. Put another way, this command will stop the motor (configured deceleration limits will be respected) and not allow the motor to start again until the Safe-Start conditions required by the Input Mode are satisfied.

6.4. Controller variables

The Simple Motor Controller G2 maintains a set of variables that contain real-time information about the controller's inputs, outputs, and state. Most of these variables are all displayed in Status tab of the Simple Motor Control Center G2 software (see **Section 3.2**), and they can all be requested via the serial or I²C interfaces (see the "Get variable" command in **Section 6.2.1** and **Section 6.3.1**). The "Get variable" command reports all variables as 16-bit (2-byte values transmitted least significant byte first), though not all variables use all 16 bits.

Status flag registers

Status flag registers are unsigned, 16-bit values whose bits convey general information about the controller's status, such as any errors that have occurred, the errors are currently stopping the motor, and sources of controller output limitations.

ID	Name	Description
0	Error status	The set bits of this variable indicate the errors that are currently stopping the motor. The motor can only be driven when this register has a value of 0. (See **Section 3.3** for error descriptions.) • **Bit 0:** Safe start violation • **Bit 1:** Required channel invalid • **Bit 2:** Serial error • **Bit 3:** Command timeout • **Bit 4:** Limit/kill switch • **Bit 5:** Low VIN • **Bit 6:** High VIN • **Bit 7:** Over temperature • **Bit 8:** Motor driver error • **Bit 9:** ERR line high • **Bits 10-15:** *reserved*
1	Errors occurred	The set bits of this register indicate the errors that have occurred since this register was last cleared. This status register has the same bit assignments as the *Error status* register documented above. **Reading this variable clears all of the bits.**
2	Serial errors occurred	The set bits of this variable indicate the serial errors that have occurred since this variable was last cleared. **Reading this variable clears all of the bits.** (See **Section 3.3** for serial error descriptions.) • **Bit 0:** *reserved* • **Bit 1:** Frame • **Bit 2:** Noise • **Bit 3:** RX overrun • **Bit 4:** Format • **Bit 5:** CRC • **Bits 6-16:** *reserved*
3	Limit status	The set bits of this variable indicate things that are currently limiting the motor controller.

		• **Bit 0:** Motor is not allowed to run due to an error or safe-start violation. • **Bit 1:** Temperature is actively reducing target speed. • **Bit 2:** Max speed limit is actively reducing target speed (target speed > max speed). • **Bit 3:** Starting speed limit is actively reducing target speed to zero (target speed < starting speed). • **Bit 4:** Motor speed is not equal to target speed because of acceleration, deceleration, or brake duration limits. • **Bit 5:** RC1 is configured as a limit/kill switch and the switch is active (scaled value ≥ 1600). • **Bit 6:** RC2 limit/kill switch is active (scaled value ≥ 1600). • **Bit 7:** AN1 limit/kill switch is active (scaled value ≥ 1600). • **Bit 8:** AN2 limit/kill switch is active (scaled value ≥ 1600). • **Bit 9:** USB kill switch is active. • **Bits 10-15:** *reserved*
127	Reset flags	Flags indicating the source of the last board reset. This variable does not change while the controller is running. You can view this information in the Device Information window of the Control Center, which is available from the Device menu, and for the first two seconds after start-up, the yellow status LED flashes a pattern that indicates the last reset source (see **Section 3.5**). • **0x04 (4):** $\overline{\text{RST}}$ pin pulled low by external source. • **0x0C (12):** Power reset (VIN got too low or was disconnected). • **0x14 (20):** Software reset (by firmware upgrade process). • **0x24 (38):** Watchdog timer reset (should never happen; this could indicate a firmware bug).

RC channel inputs

The raw and scaled signals measured on the RC channel inputs are always available through serial/I²C variable requests, which allows programs using the serial interface to factor the channel inputs into their motor control algorithms. If no valid signal is detected, the raw channel value is reported as 0xFFFF (65535) and the scaled channel value is reported as 0. The Simple Motor Controller G2 is always reading the RC input channels, even when the Input Mode is not RC.

ID	Name	Type	Description	Units
4	RC1 unlimited raw value	unsigned 16-bit	The positive pulse width of the signal on RC channel 1. This value is 0xFFFF (65535) if no valid signal is detected.	0.25 µs
5	RC1 raw value	unsigned 16-bit	The positive pulse width of the signal on RC channel 1. This value is 0xFFFF (65535) if no valid signal is detected or if the signal is outside of the error max/error min channel calibration settings.	0.25 µs
6	RC1 scaled value	signed 16-bit	The scaled version of the RC1 raw value (based on RC channel 1 calibration settings). This value is 0 if the raw value is 0xFFFF, else it ranges from −3200 to +3200.	internal units
8	RC2 unlimited raw value	unsigned 16-bit	See RC1 unlimited raw value.	0.25 µs
9	RC2 raw value	unsigned 16-bit	See RC1 raw value.	0.25 µs
10	RC2 scaled value	signed 16-bit	See RC1 scaled value.	internal units

Analog channel inputs

The raw and scaled voltages measured on the analog channel inputs are always available through serial/I²C variable requests, which allows programs using the serial interface to factor the channel inputs into their motor control algorithms. If the controller detects a disconnected potentiometer (this requires potentiometer disconnect detection to be enabled under the Advanced Settings tab), the raw channel value is reported as 0xFFFF (65535) and the scaled channel value is reported as 0. The Simple Motor Controller G2 is always reading the analog input channels, even when the input mode is not "Analog".

ID	Name	Type	Description	Units
12	AN1 unlimited raw value	unsigned 16-bit	The 12-bit ADC reading of analog channel 1. This value is 0xFFFF (65535) if the controller detects the input is disconnected.	0=0 V, 4095=3.3 V
13	AN1 raw value	unsigned 16-bit	The 12-bit ADC reading of analog channel 1. This value is 0xFFFF (65535) if the controller detects the input is disconnected or if the signal is outside of the error max/ error min channel calibration settings.	0=0 V, 4095=3.3 V
14	AN1 scaled value	signed 16-bit	The scaled version of the AN1 raw value (based on analog channel 1 calibration settings). This value is 0 if the raw value is 0xFFFF, else it ranges from −3200 to +3200.	internal units
16	AN2 unlimited raw value	unsigned 16-bit	See AN1 unlimited raw value.	0=0 V, 4095=3.3 V
17	AN2 raw value	unsigned 16-bit	See AN1 raw value.	0=0 V, 4095=3.3 V
18	AN2 scaled value	signed 16-bit	See AN1 scaled value.	internal units

Diagnostic variables

The following variables can be used to monitor various internal conditions of the Simple Motor Controller G2, such as the input voltage, the board temperature, the and the motor speed.

ID	Name	Type	Description	Units
20	Target speed	signed 16-bit	Motor target speed (−3200 to +3200) requested by the controlling interface.	internal units
21	Speed	signed 16-bit	Current speed of the motor (−3200 to +3200).	internal units
22	Brake amount	unsigned 16-bit	When speed=0, this variable indicates whether the controller is braking or not. A value of 0 indicates coasting, and a value of 32 indicates braking. Otherwise, it has a value of 0xFF (255). The high byte of this variable is always zero.	0=coast, 32=brake
23	Input voltage	unsigned 16-bit	Measured voltage on the VIN pin.	mV
24	Temperature A	unsigned 16-bit	Board temperature. Temperatures below freezing are reported as 0. Errors measuring the temperature are reported as 3000 (300 °C).	0.1 °C
25	Temperature B	unsigned 16-bit	Board temperature measured at a different location. See temperature A, documented above.	0.1 °C
26	RC period	unsigned 16-bit	If there is a valid signal on RC1, this variable contains the signal period. Otherwise, this variable has a value of 0.	0.1 ms
27	Baud rate register	unsigned 16-bit	Value of the controller's baud rate register (BRR). Convert to units of bps with the equation `72,000,000/BRR`. In automatic baud detection mode, BRR has a value of 0 until the controller has detected the baud rate.	seconds per 7.2e7 bits
28	Up time (low)	unsigned 16-bit	Two lower bytes of the number of milliseconds that have elapsed since the controller was last reset or powered up.	ms
29	Up time (high)	unsigned 16-bit	Two upper bytes of the number of milliseconds that have elapsed since the controller was last reset or powered up.	65,536 ms

Motor limits

These variables contain the user-imposed limits on the motor output, such as maximum speed, acceleration, and deceleration. These variables are initialized to the hard motor limit settings (see

Section 5.2) every time the controller is powered up or reset and every time the "Apply settings" button is pressed in the Simple Motor Control Center G2. Most of these limits can be changed while the controller is running to impose stricter/safer limits than the hard motor limit settings (see the Set Motor Limit command in **Section 6.2.1** and **Section 6.3.1**).

ID	Name	Type	Description	Units
30	Max speed forward	unsigned 16-bit	Maximum allowed motor speed in the forward direction (0 to 3200).	internal units
31	Max acceleration forward	unsigned 16-bit	Maximum allowed motor acceleration in the forward direction (0 to 3200; 0 means no limit).	Δ*speed* per *update period*
32	Max deceleration forward	unsigned 16-bit	Maximum allowed motor deceleration from the forward direction (0 to 3200; 0 means no limit).	Δ*speed* per *update period*
33	Brake duration forward	unsigned 16-bit	Time spent braking (at speed 0) when transitioning from forward to reverse.	ms
34	Starting speed forward	unsigned 16-bit	Minimum allowed motor speed in the forward direction (0 to 3200).	internal units
36	Max speed reverse	unsigned 16-bit	Maximum allowed motor speed in the reverse direction (0 to 3200).	internal units
37	Max acceleration reverse	unsigned 16-bit	Maximum allowed motor acceleration in the reverse direction (0 to 3200; 0 means no limit).	Δ*speed* per *update period*
38	Max deceleration reverse	unsigned 16-bit	Maximum allowed motor deceleration from the reverse direction (0 to 3200; 0 means no limit).	Δ*speed* per *update period*
39	Brake duration reverse	unsigned 16-bit	Time spent braking (at speed 0) when transitioning from reverse to forward.	ms
40	Starting speed reverse	unsigned 16-bit	Minimum allowed motor speed in the reverse direction (0 to 3200).	internal units

Current limiting and measurement

ID	Name	Type	Description	Units
42	Current limit	unsigned 16-bit	The hardware current limit currently being used. See **Section 5.2**.	internal units
43	Raw current	unsigned 16-bit	The raw motor current measurement. See **Section 5.2**.	internal units
44	Current	unsigned 16-bit	Measurement of the motor current in milliamps.	mA
45	Current limiting consecutive count	unsigned 16-bit	The number of consecutive 10 ms time periods in which the hardware current limiting has activated.	count
46	Current limiting occurrence count	unsigned 16-bit	The number of 10 ms time periods in which the hardware current limit has activated since the last time this variable was cleared. **Reading this variable clears it, resetting it to 0.**	count

6.5. Cyclic redundancy check (CRC) error detection

For certain applications, verifying the integrity of the data you are sending and receiving can be very important. Because of this, the Simple Motor Controller G2 has optional 7-bit cyclic redundancy checking, which is similar to a checksum but more robust as it can detect errors that would not affect a checksum, such as an extra zero byte or bytes out of order.

Cyclic redundancy checking can be enabled in the "Input settings" tab of the Simple Motor Control Center G2. If CRC is enabled for commands, the Simple Motor Controller G2 expects an extra byte to be added onto the end of every binary mode command packet (CRC error checking is not available for ASCII commands). The most-significant bit of this byte must be cleared, and the seven least-significant bits must be the 7-bit CRC for that packet. If this CRC byte is incorrect, a CRC error will occur and the command will be ignored. The Simple Motor Controller G2 will append a CRC byte to the data it transmits in response to serial commands if CRC is enabled for responses.

A detailed account of how cyclic redundancy checking works is beyond the scope of this document, but you can find more information using **Wikipedia [http://en.wikipedia.org/wiki/Cyclic_redundancy_check]**. The 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 Simple Motor Controller G2 uses CRC-7, which means it uses an 8-bit polynomial 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.

For sample C code that computes the CRC byte of a command packet, see **Section 8.13**.

The CRC implemented on the Simple Motor Controller G2 is the same as the one on the original Simple Motor Controllers, the **Maestro** **[https://www.pololu.com/product/1352]** servo controllers, **Jrk** **[https://www.pololu.com/product/3142]** motor controllers, and **qik** **[https://www.pololu.com/product/1110]** motor controllers, but it differs from that on the **TReX** **[https://www.pololu.com/product/777]** motor controller. Instead of being done MSB first, the computation is performed 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:

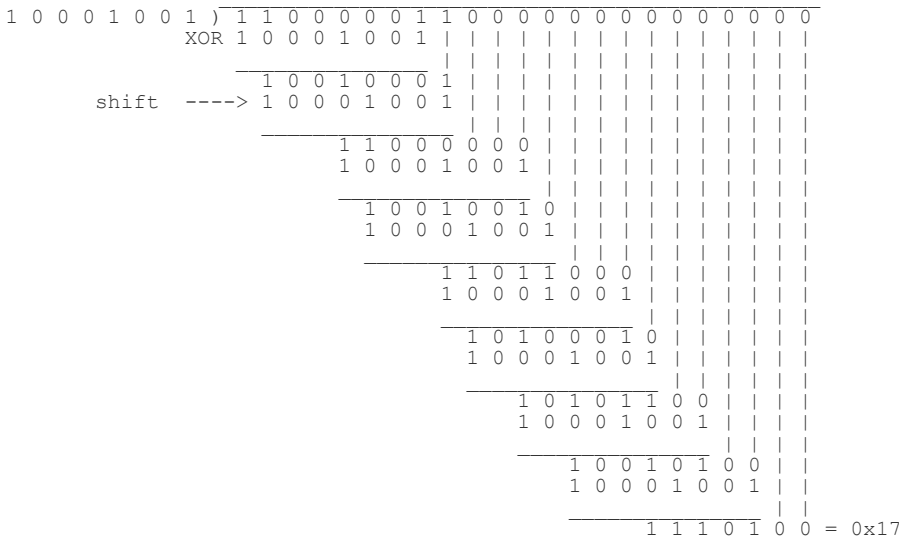
1. Express your 8-bit CRC-7 polynomial and message in binary, LSB first. The polynomial **0x91** is written as **10001001**.
2. Add 7 zeros to the end of your message.
3. Write your CRC-7 polynomial underneath the message so that the LSB of your polynomial is directly below the LSB of your message.
4. 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 Simple Motor Controller with CRC enabled).

If you have never encountered CRCs before, this probably sounds a lot more complicated than it really is. The following example shows that the CRC-7 calculation is not that difficult. For the example, we will use a two-byte sequence: **0x83, 0x01**.

Steps 1 & 2 (write as binary, least significant bit first, 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
```

Steps 3, 4, & 5:



So the full command packet we would send with CRC enabled is: **0x83, 0x01, 0x17**.

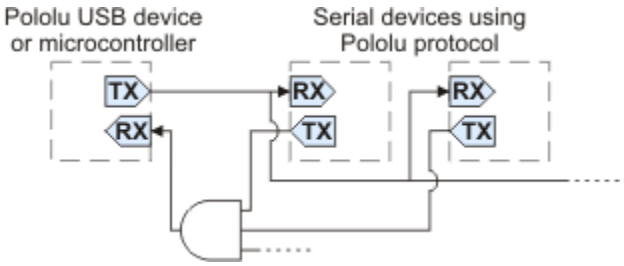
6.6. Serial daisy chaining

This section is a guide to integrating the Simple Motor Controller G2 into a project that has multiple TTL serial devices that use a compatible protocol.

First of all, you will need to decide whether to use the Pololu protocol, the Mini SSC protocol, or a mix of both (see **Section 6.2**). You must make sure that no serial command you send will cause unintended operations on the devices it was not addressed to. If you want to daisy chain several G2 Simple Motor Controllers together, you can use a mixture of both protocols. If you want to daisy chain the Simple Motor Controller G2 with other devices that use the Pololu protocol, you can use the Pololu protocol. If you want to daisy chain the Simple Motor Controller G2 with other devices that use the Mini SSC protocol, you can use the Mini SSC protocol.

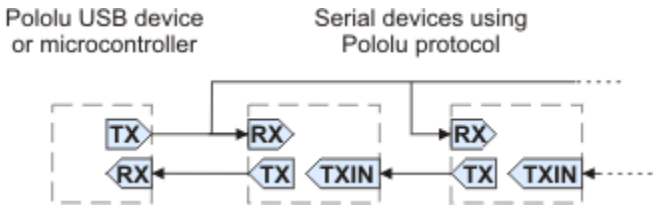
Secondly, assign each device in the project a different device number so that they can be individually addressed by your serial commands. For the Simple Motor Controller G2, this can be done in the "Input settings" tab of the Simple Motor Control Center G2.

The following diagram shows how to connect one master and many slave devices together into a chain. Each of the slave devices may be a Simple Motor Controller G2 or any other TTL serial device, such as a **Maestro [https://www.pololu.com/product/1352]**, **Jrk [https://www.pololu.com/product/3142]**, **qik [https://www.pololu.com/product/1110]** or other microcontroller.



Daisy chaining serial devices using the Pololu protocol. An optional AND gate is used to join multiple TX lines.

The Simple Motor Controller G2 has a special input called **TXIN** that eliminates the need for an external AND gate (the AND gate is built in to the control.) To make a chain of devices using the **TXIN** input, connect them like this:



Daisy chaining serial devices that have a TXIN input.

For additional connection diagrams and more information about the **TXIN** pin, see **Section 4.2**.

Connections

Connect the TX line of your controlling device to the RX lines of all of the slave devices. Sent commands will then be received by all slaves.

When receiving serial responses from multiple slaves, each device should only transmit when requested, so if each device is addressed separately, multiple devices will not transmit simultaneously. However, the TX outputs are driven high when not sending data, so they cannot simply be wired together. Instead, you can use an AND gate, as shown in the diagram, to combine the signals, or you can use the TXIN pin as described above if the device has one. Note that in many cases receiving responses is not necessary, and the TX lines can be left unconnected.

Whenever connecting devices, remember to wire the grounds together, and ensure that each device is properly powered. Unpowered devices with a TTL serial port can turn on or partially on, drawing power from the serial line, which means that extra care must be taken when turning power off and on to reset the devices.

Sending commands

The Pololu Protocol or Mini SSC protocol should be used when multiple Pololu devices are receiving the same serial data. This allows the devices to be individually addressed, and it allows responses to be sent without collisions.

If the devices are configured to detect the baud rate, then when you issue your first Pololu Protocol command, the devices can automatically detect the baud from the initial 0xAA byte.

Some older Pololu devices use 0x80 as an initial command byte. If you want to chain these together with devices expecting 0xAA, 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 Simple Motor Controller can detect the baud rate. Once all devices have detected the baud rate, Pololu devices that expect a leading command byte of 0x80 will ignore command packets that start with 0xAA, and Pololu devices that use the Pololu Protocol, such as the Simple Motor Controller, will ignore command packets that start with 0x80.

7. Writing PC software to control the Simple Motor Controller G2

There are two ways to write PC software to control a Simple Motor Controller G2 that is connected via USB: you can use the native USB interface and the USB virtual serial port. The native USB interface provides more features than the serial port, such as the ability to change settings and select the Simple Motor Controller G2 by its serial number. Also, the USB interface allows you to recover more easily from temporary disconnections. The virtual serial port interface is often easier to use to get started with because of its simplicity and because of the availability of serial port libraries in many programming languages.

Native USB Interface

The simplest way to use the native USB interface is to write a program in the language of your choice that invokes smcg2cmd, the command-line utility that comes with the Simple Motor Controller G2 software. See **Section 8** for example code that runs smcg2cmd.

For some examples that access the native USB interface more directly, see **Section 8.3**.

USB virtual serial port

Almost any programming language is capable of accessing the Simple Motor Controller G2's USB virtual serial port. One option is the Microsoft .NET framework, which is free to use and contains a SerialPort class that makes it easy to read and write bytes from a serial port. Also, see **Section 8** for example code that works with the USB virtual serial port.

8. Example code

8.1. Example code to run smcg2cmd in C

The example C code below shows how to invoke the Simple Motor Controller G2 Command-line Utility (smcg2cmd) to control a Simple Motor Controller G2 via USB.

If you have multiple Simple Motor Controller G2 devices connected to your computer via USB, you will need to use the `-d` option to specify the serial number of the device you want to use. For example, to set the speed to 2400 on a Simple Motor Controller G2 with serial number 0012-3456-789A-BCDE-F012-3456, you can run the command `smcg2cmd -d 0012-3456-789A-BCDE-F012-3456 --resume --speed 2400`. You can run `smcg2cmd --list` in a shell to get the serial numbers of all the connected Simple Motor Controller G2 devices.

In the example below, the child smcg2cmd process uses the same error pipe as the parent example program, so you will see any error messages printed by smcg2cmd if you run the example program in a terminal.

```c
// Uses smcg2cmd to set the target speed of the Simple Motor Controlelr G2
// over USB.
//
// NOTE: The SMC G2's input mode must be "Serial / USB".

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

// Runs the given shell command.  Returns 0 on success, -1 on failure.
int run_command(const char * command)
{
  int result = system(command);
  if (result)
  {
    fprintf(stderr, "Command failed with code %d: %s\n", result, command);
    return -1;
  }
  return 0;
}

// Sends the "Exit safe start" command and also sets the target speed.
// Returns 0 on success and -1 on failure.
int smc_set_target_speed(int16_t speed)
{
  char command[1024];
  snprintf(command, sizeof(command),
    "smcg2cmd --resume --speed %d", speed);
  return run_command(command);
}

int main()
{
  printf("Setting target speed to 2400.\n");
  int result = smc_set_target_speed(2400);
  if (result) { return 1; }
  return 0;
}
```

8.2. Example code to run smcg2cmd in Python

The example Python code below shows how to invoke the Simple Motor Controller G2 Command-line Utility (smcg2cmd) to send and receive data from a Simple Motor Controller G2 via USB. It demonstrates how to set the target speed of the controller using the `--speed` option and how to read variables using the `-s` option. This code works with either Python 2 or Python 3.

If you have multiple Simple Motor Controller G2 devices connected to your computer via USB, you will need to use the `-d` option to specify the serial number of the device you want to use. For example, to set the speed to 2400 on a Simple Motor Controller G2 with serial number 0012-3456-789A-BCDE-F012-3456, you can run the command `smcg2cmd -d 0012-3456-789A-BCDE-F012-3456 --resume --speed 2400`. You can run `smcg2cmd --list` in a shell to get the serial numbers of all the connected Simple Motor Controller G2 devices.

In the example below, the child smcg2cmd process uses the same error pipe as the Python process,

so you will see any error messages printed by smcg2cmd if you run the Python program in a terminal. Additionally, if there is an error, Python's `subprocess.check_output` method will detect it (by checking the smcg2cmd process exit status) and raise an exception.

```
# Uses smcg2cmd to control a Simple Motor Controller G2 over USB.
# Works with either Python 2 or Python 3.
#
# NOTE: The Simple Motor Controller's input mode must be "Serial / USB".

import subprocess
import re

def smcg2cmd(*args):
  return subprocess.check_output(['smcg2cmd'] + list(args))

def get_target_speed_from_status(status):
  e = re.compile('^Target Speed:\s*(-?\d+)\s*\r$', re.IGNORECASE | re.MULTILINE)
  r = e.search(status)
  if r is None:
    raise RuntimeError('Failed to parse status')
  return int(r.group(1))

status = smcg2cmd('-s').decode()

target_speed = get_target_speed_from_status(status)
print("Target speed is {}.".format(target_speed))

new_speed = 3200 if target_speed <= 0 else -3200
print("Setting target speed to {}.".format(new_speed))
smcg2cmd('--resume', '--speed', str(new_speed))
```

8.3. Example native USB code in C#, Visual C++, and VB .NET

The **Pololu USB Software Development Kit [https://www.pololu.com/docs/0J41]** supports Windows and Linux, and includes the source code for:

- **SmcG2Example1**: a simple example application that uses native USB and has three buttons for setting the motor speed. Versions of this example are available in C#, Visual Basic .NET, and Visual C++.
- **SmcG2Example2**: an example graphical application that has a scrollbar for setting the motor speed over native USB (written in C#).
- **SmcG2Cmd**: the command-line utility for configuring and controlling the Simple Motor Controller G2 (written in C#).
- **SmcG2**: A .NET class library that enables native USB communication with the Simple Motor Controller G2 (written in C#).

You can modify the applications in the SDK to suit your needs or you can use the class library to integrate the Simple Motor Controller G2 in to your own applications.

8.4. Example serial code for Arduino

Arduino R3, top view.

This section has two example programs for communicating with the Simple Motor Controller G2's TTL serial interface from an **Arduino** **[https://www.pololu.com/category/125/arduino]**, **A-Star** **[https://www.pololu.com/category/149/a-star-programmable-controllers]**, or other Arduino-compatible controller.

The **Arduino Uno** **[https://www.pololu.com/product/2191]** uses its hardware serial (or "UART") lines for programming and for debugging with the Arduino IDE's serial monitor, so we do not recommend using these lines to communicate with peripheral serial devices like the Simple Motor Controller G2. Instead, we recommend using the **SoftwareSerial** **[http://arduino.cc/hu/Reference/SoftwareSerial]** library included with the Arduino IDE, which lets you use arbitrary I/O lines for transmitting and receiving serial bytes. The drawback is that software serial requires much more processing time than hardware serial.

In the following examples, we use the SoftwareSerial library to transmit bytes on digital pin 4 and receive bytes on digital pin 3.

These sample programs require the Simple Motor Controller G2 to have a **fixed baud rate set to 19200 bps**. It must also be in **binary serial mode** with the CRC disabled. Auto baud rate detection can be used, but it is not recommended because of inaccuracy in the SoftwareSerial library.

Simple example

This example assumes the following connections exist between the Arduino and the Simple Motor Controller G2:

- Arduino digital pin **4** to Simple Motor Controller **RX**
- Arduino **GND** to Simple Motor Controller **GND**

See **Section 4.3** for more information on connecting a serial device to the Simple Motor Controller.

This program demonstrates how to initiate serial communication with the Simple Motor Controller G2 and how to send commands to set the motor speed. For information about the serial commands used by this sample code, refer to **Section 6.2.1**. Note that the Simple Motor Controller G2 must be powered when this Arduino sketch starts running.

```
#include <SoftwareSerial.h>
#define rxPin 3  // pin 3 connects to smcSerial TX  (not used in this example)
#define txPin 4  // pin 4 connects to smcSerial RX
SoftwareSerial smcSerial = SoftwareSerial(rxPin, txPin);

// required to allow motors to move
// must be called when controller restarts and after any error
void exitSafeStart()
{
  smcSerial.write(0x83);
}

// speed should be a number from -3200 to 3200
void setMotorSpeed(int speed)
{
  if (speed < 0)
  {
    smcSerial.write(0x86);  // motor reverse command
    speed = -speed;  // make speed positive
  }
  else
  {
    smcSerial.write(0x85);  // motor forward command
  }
  smcSerial.write(speed & 0x1F);
  smcSerial.write(speed >> 5 & 0x7F);
}

void setup()
{
  // Initialize software serial object with baud rate of 19.2 kbps.
  smcSerial.begin(19200);

  // The Simple Motor Controller must be running for at least 1 ms
  // before we try to send serial data, so we delay here for 5 ms.
  delay(5);

  // If the Simple Motor Controller has automatic baud detection
  // enabled, we first need to send it the byte 0xAA (170 in decimal)
  // so that it can learn the baud rate.
  smcSerial.write(0xAA);

  // Next we need to send the Exit Safe Start command, which
  // clears the safe-start violation and lets the motor run.
  exitSafeStart();
}

void loop()
{
  setMotorSpeed(3200);  // full-speed forward
  delay(1000);
  setMotorSpeed(-3200);  // full-speed reverse
  delay(1000);
}
```

Advanced Example

This example assumes the following connections exist between the Arduino and the Simple Motor Controller G2:

- Arduino digital pin **3** to Simple Motor Controller **TX**
- Arduino digital pin **4** to Simple Motor Controller **RX**
- Arduino digital pin **5** to Simple Motor Controller **$\overline{\text{RST}}$**
- Arduino digital pin **6** to Simple Motor Controller **ERR**
- Arduino **GND** to Simple Motor Controller **GND**

This program demonstrates how to initiate serial communication with the Simple Motor Controller G2 and how to send commands to set the motor speed, read variables, and change the temporary motor limits. For information about the serial commands used by this sample code, refer to **Section 6.2.1**. It will be more interesting if you have input power and a motor connected to your Simple Motor Controller G2 (see **Section 4.1**), but you can see some interesting things even without a motor connected by using the Status tab of the Simple Motor Control Center G2 application to monitor the effect this sketch has on the controller's variables (see **Section 3.2**).

```
#include <SoftwareSerial.h>
#define rxPin 3    // pin 3 connects to SMC TX
#define txPin 4    // pin 4 connects to SMC RX
#define resetPin 5 // pin 5 connects to SMC nRST
#define errPin 6   // pin 6 connects to SMC ERR
SoftwareSerial smcSerial = SoftwareSerial(rxPin, txPin);

// some variable IDs
#define ERROR_STATUS 0
#define LIMIT_STATUS 3
#define TARGET_SPEED 20
#define INPUT_VOLTAGE 23
#define TEMPERATURE 24

// some motor limit IDs
#define FORWARD_ACCELERATION 5
#define REVERSE_ACCELERATION 9
#define DECELERATION 2

// read a serial byte (returns -1 if nothing received after the timeout expires)
int readByte()
{
  char c;
  if(smcSerial.readBytes(&c, 1) == 0){ return -1; }
  return (byte)c;
}

// required to allow motors to move
// must be called when controller restarts and after any error
void exitSafeStart()
{
  smcSerial.write(0x83);
}

// speed should be a number from -3200 to 3200
void setMotorSpeed(int speed)
{
  if (speed < 0)
  {
    smcSerial.write(0x86);  // motor reverse command
    speed = -speed;  // make speed positive
  }
  else
  {
    smcSerial.write(0x85);  // motor forward command
  }
  smcSerial.write(speed & 0x1F);
  smcSerial.write(speed >> 5 & 0x7F);
}

unsigned char setMotorLimit(unsigned char  limitID, unsigned int limitValue)
{
  smcSerial.write(0xA2);
  smcSerial.write(limitID);
  smcSerial.write(limitValue & 0x7F);
  smcSerial.write(limitValue >> 7);
  return readByte();
}

// returns the specified variable as an unsigned integer.
// if the requested variable is signed, the value returned by this function
// should be typecast as an int.
```

```
unsigned int getVariable(unsigned char variableID)
{
  smcSerial.write(0xA1);
  smcSerial.write(variableID);
  return readByte() + 256 * readByte();
}

void setup()
{
  Serial.begin(115200);    // for debugging (optional)
  smcSerial.begin(19200);

  // briefly reset SMC when Arduino starts up (optional)
  pinMode(resetPin, OUTPUT);
  digitalWrite(resetPin, LOW);  // reset SMC
  delay(1);  // wait 1 ms
  pinMode(resetPin, INPUT);  // let SMC run again

  // must wait at least 1 ms after reset before transmitting
  delay(5);

  // this lets us read the state of the SMC ERR pin (optional)
  pinMode(errPin, INPUT);

  smcSerial.write(0xAA);  // send baud-indicator byte
  setMotorLimit(FORWARD_ACCELERATION, 4);
  setMotorLimit(REVERSE_ACCELERATION, 10);
  setMotorLimit(DECELERATION, 20);
  // clear the safe-start violation and let the motor run
  exitSafeStart();
}

void loop()
{
  setMotorSpeed(3200);  // full-speed forward
  // signed variables must be cast to ints:
  Serial.println((int)getVariable(TARGET_SPEED));
  delay(1000);
  setMotorSpeed(-3200);  // full-speed reverse
  Serial.println((int)getVariable(TARGET_SPEED));
  delay(1000);

  // write input voltage (in millivolts) to the serial monitor
  Serial.print("VIN = ");
  Serial.print(getVariable(INPUT_VOLTAGE));
  Serial.println(" mV");

  // if an error is stopping the motor, write the error status variable
  // and try to re-enable the motor
  if (digitalRead(errPin) == HIGH)
  {
    Serial.print("Error Status: 0x");
    Serial.println(getVariable(ERROR_STATUS), HEX);
    // once all other errors have been fixed,
    // this lets the motors run again
    exitSafeStart();
  }
}
```

8.5. Example serial code for Orangutan

The **Orangutan robot controllers** **[https://www.pololu.com/category/8/robot-controllers]** feature user-programmable Atmel AVR microcontrollers interfaced with additional hardware useful for controlling robots. They are programmable in C or C++ and supported by the **Pololu AVR library** **[https://www.pololu.com/docs/0J20]**, which makes it easy to use the integrated hardware and AVR peripherals, such as the UART module. Unlike the Arduino, the hardware serial lines are completely available on the Orangutans, so software serial is not necessary when connecting to serial devices like the Simple Motor Controller G2.

In the following example programs, we use the OrangutanSerial functions from the Pololu AVR library to transmit bytes on pin PD1. In the advanced example, we use the OrangutanSerial functions to receive bytes on pin PD0, and we use the OrangutanLCD functions to report feedback obtained from the Simple Motor Controller G2. See the **Pololu AVR library command reference** **[https://www.pololu.com/docs/0J18]** for more information on these functions.

This code requires the Simple Motor Controller to have **automatic baud rate detection enabled** or to have a **fixed baud rate set to 115200 bps**. It must also be in **Binary serial mode** with the CRC disabled.

Simple example

This example assumes the following connections exist between the Orangutan and the Simple Motor Controller:

- Orangutan pin **PD0** to Simple Motor Controller **TX**
- Orangutan **GND** to Simple Motor Controller **GND**

Pin PD0 is the Orangutan's hardware serial receive line and must be connected to the Simple Motor Controller as described above for this sample program to work. See **Section 4.3** for more information on connecting a serial device to the Simple Motor Controller G2.

This program demonstrates how to initiate serial communication with the Simple Motor Controller and how to send commands to set the motor speed. For information about the serial commands used by this sample code, refer to **Section 6.2.1**. Note that the Simple Motor Controller must be powered when this Orangutan program starts running.

```
#include <pololu/orangutan.h>

char command[3];

// These first two functions call the appropriate Pololu AVR library serial functions
// depending on which Orangutan you are using.  The Orangutan SVP and X2 have multiple
// serial ports, so the serial functions for these devices require an extra argument
// specifying which port to use.  You can simplify this program by just calling the
// library function appropriate for your Orangutan board.

void setBaudRate(unsigned long baud)
{
#if _SERIAL_PORTS > 1  // Orangutan X2 and SVP users
  serial_set_baud_rate(UART0, baud);
#else
  serial_set_baud_rate(baud);
#endif
}

void sendBlocking(char * buffer, unsigned char size)
{
#if _SERIAL_PORTS > 1  // Orangutan X2 and SVP users
  serial_send_blocking(UART0, buffer, size);
#else
  serial_send_blocking(buffer, size);
#endif
}

// required to allow motors to move
// must be called when controller restarts and after any error
void exitSafeStart()
{
  command[0] = 0x83;
  sendBlocking(command, 1);
}

// speed should be a number from -3200 to 3200
void setMotorSpeed(int speed)
{
  if (speed < 0)
  {
    command[0] = 0x86;  // motor reverse command
    speed = -speed;  // make speed positive
  }
  else
  {
    command[0] = 0x85;  // motor forward command
  }
  command[1] = speed & 0x1F;
  command[2] = speed >> 5;
  sendBlocking(command, 3);
}

// initialization code called once when the program starts running
void setup()
{
  // initialize hardware serial (UART0) with baud rate of 115.2 kbps
  setBaudRate(115200);

  // the Simple Motor Controller must be running for at least 1 ms
  // before we try to send serial data, so we delay here for 5 ms
  delay_ms(5);
```

```
63
64    // if the Simple Motor Controller has automatic baud detection
65    // enabled, we first need to send it the byte 0xAA (170 in decimal)
66    // so that it can learn the baud rate
67    command[0] = 0xAA;
68    sendBlocking(command, 1);  // send baud-indicator byte
69
70    // next we need to send the Exit Safe Start command, which
71    // clears the safe-start violation and lets the motor run
72    exitSafeStart();  // clear the safe-start violation and let the motor run
73  }
74
75  // program execution starts here
76  int main()
77  {
78    setup();
79    while (1)  // loop forever
80    {
81      setMotorSpeed(3200);
82      delay_ms(1000);
83      setMotorSpeed(-3200);
84      delay_ms(1000);
85    }
86  }
```

Advanced Example

This example assumes the following connections exist between the Orangutan and the Simple Motor Controller:

- Orangutan pin **PD0** to Simple Motor Controller **TX**
- Orangutan pin **PD1** to Simple Motor Controller **RX**
- Orangutan pin **PC0** to Simple Motor Controller **$\overline{\text{RST}}$**
- Orangutan pin **PC1** to Simple Motor Controller **ERR**
- Orangutan **GND** to Simple Motor Controller **GND**

Pins PD0 and PD1 are the Orangutan's hardware serial receive and transmit lines, respectively, and must be connected to the Simple Motor Controller as described above for this sample program to work. There is nothing special about pins PC0 and PC1, however; you can connect any free digital pins to the Simple Motor Controller $\overline{\text{RST}}$ and ERR pins if you change the pin definitions at the top of the sample program accordingly. See **Section 4.3** for more information on connecting a serial device to the Simple Motor Controller.

This program demonstrates how to initiate serial communication with the Simple Motor Controller and how to send commands to set the motor speed, read variables, and change the temporary motor limits. For information about the serial commands used by this sample code, refer to **Section 6.2.1**. It will be more interesting if you have input power and a motor connected to your Simple Motor Controller (see **Section 4.1**), but you can see some interesting things even without a motor connected by using the

Status tab of the Simple Motor Control Center G2 application to monitor the effect this program has on the controller's variables (see **Section 3.2**).

```
#include <pololu/orangutan.h>
#define resetPin IO_C0  // pin PC0 connects to SMC nRST
#define errPin IO_C1  // pin PC1 connects to SMC ERR

// some variable IDs
#define ERROR_STATUS 0
#define LIMIT_STATUS 3
#define TARGET_SPEED 20
#define INPUT_VOLTAGE 23
#define TEMPERATURE 24

// some motor limit IDs
#define FORWARD_ACCELERATION 5
#define REVERSE_ACCELERATION 9
#define DECELERATION 2

char command[4];

// These first three functions call the appropriate Pololu AVR library serial functions
// depending on which Orangutan you are using.  The Orangutan SVP and X2 have multiple
// serial ports, so the serial functions for these devices require an extra argument
// specifying which port to use.  You can simplify this program by just calling the
// library function appropriate for your Orangutan board.

void setBaudRate(unsigned long baud)
{
#if _SERIAL_PORTS > 1  // Orangutan X2 and SVP users
  serial_set_baud_rate(UART0, baud);
#else
  serial_set_baud_rate(baud);
#endif
}

void sendBlocking(char * buffer, unsigned char size)
{
#if _SERIAL_PORTS > 1  // Orangutan X2 and SVP users
  serial_send_blocking(UART0, buffer, size);
#else
  serial_send_blocking(buffer, size);
#endif
}

char receiveBlocking(char * buffer, unsigned char size, unsigned int timeout_ms)
{
#if _SERIAL_PORTS > 1  // Orangutan X2 and SVP users
  return serial_receive_blocking(UART0, buffer, size, timeout_ms);
#else
  return serial_receive_blocking(buffer, size, timeout_ms);
#endif
}

// required to allow motors to move
// must be called when controller restarts and after any error
void exitSafeStart()
{
  command[0] = 0x83;
  sendBlocking(command, 1);
}

// speed should be a number from -3200 to 3200
void setMotorSpeed(int speed)
{
```

```
  if (speed < 0)
  {
    command[0] = 0x86;  // motor reverse command
    speed = -speed;  // make speed positive
  }
  else
  {
    command[0] = 0x85;  // motor forward command
  }
  command[1] = speed & 0x1F;
  command[2] = speed >> 5;
  sendBlocking(command, 3);
}

char setMotorLimit(unsigned char  limitID, unsigned int limitValue)
{
  command[0] = 0xA2;
  command[1] = limitID;
  command[2] = limitValue & 0x7F;
  command[3] = limitValue >> 7;
  sendBlocking(command, 4);

  char response = -1;
  receiveBlocking(&response, 1, 500);
  return response;
}

// returns the specified variable as an unsigned integer.
// if the requested variable is signed, the value returned by this function
// should be typecast as an int.
unsigned int getVariable(unsigned char variableID)
{
  command[0] = 0xA1;
  command[1] = variableID;
  sendBlocking(command, 2);

  unsigned int response;
  if (receiveBlocking((char *)&response, 2, 500))
    return 0;  // if we don't get a response in 500 ms, return 0
  return response;
}

// initialization code called once when the program starts running
void setup()
{
  setBaudRate(115200);

  // briefly reset SMC when Arduino starts up (optional)
  set_digital_output(resetPin, LOW);
  delay_ms(1);  // wait 1 ms
  set_digital_input(resetPin, HIGH_IMPEDANCE);  // let SMC run again

  // must wait at least 1 ms after reset before transmitting
  delay_ms(5);

  // this lets us read the state of the SMC ERR pin (optional)
  set_digital_input(errPin, HIGH_IMPEDANCE);

  command[0] = 0xAA;
  sendBlocking(command, 1);  // send baud-indicator byte
  setMotorLimit(FORWARD_ACCELERATION, 4);
  setMotorLimit(REVERSE_ACCELERATION, 10);
```

```
  setMotorLimit(DECELERATION, 20);
  // clear the safe-start violation and let the motor run
  exitSafeStart();
}

// main loop of the program; this executes over and over while the program runs
void loop()
{
  static int speed = 3200;  // full-speed forward

  setMotorSpeed(speed);
  speed = -speed;  // switch motor direction

  clear();  // clear the LCD and move cursor to start of first row
  print("ts=");
  // signed variables must be cast to ints:
  print_long((int)getVariable(TARGET_SPEED));
  lcd_goto_xy(0, 1);  // move LCD cursor to start of second row

  if (is_digital_input_high(errPin))
  {
    // if an error is stopping the motor, print the error status variable
    // in hex and try to re-enable the motor
    print("Err=");
    print_hex(getVariable(ERROR_STATUS));
    // once all other errors have been fixed, this lets the motor run again
    exitSafeStart();
  }
  else
  {
    // print input voltage (in Volts) to the LCD
    print("VIN=");
    unsigned int vin = getVariable(INPUT_VOLTAGE);
    // print truncated whole number of Volts
    print_unsigned_long(vin/1000);
    print_character('.');
    // print rounded tenths of a Volt
    print_unsigned_long(((vin%1000) + 50) / 100);
  }

  delay_ms(1000);
}

// program execution starts here
int main()
{
  setup();
  while (1)
  {
    loop();
  }
}
```

8.6. Example serial code for Linux and macOS in C

The example C code below works on Linux and macOS. It demonstrates how to use the USB virtual serial port or the TTL serial port to get the error status from the controller, read a variable, and set the target speed.

For this example to work, the Simple Motor Controller G2's input mode must be **Serial/USB**, the serial mode must be **Binary**, and the CRC must be disabled. These are the default settings that the controller is shipped with. If you are using TTL serial, you should set the controller to use a fixed baud rate of 9600, or change the baud rate in the code below to match whatever fixed baud rate you choose.

```c
// Uses POSIX functions to send and receive data from a
// Simple Motor Controller G2.
// NOTE: The Simple Motor Controller's input mode must be set to Serial/USB.
// NOTE: You must change the 'const char * device' line below.

#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>
#include <termios.h>

// Opens the specified serial port, sets it up for binary communication,
// configures its read timeouts, and sets its baud rate.
// Returns a non-negative file descriptor on success, or -1 on failure.
int open_serial_port(const char * device, uint32_t baud_rate)
{
  int fd = open(device, O_RDWR | O_NOCTTY);
  if (fd == -1)
  {
    perror(device);
    return -1;
  }

  // Flush away any bytes previously read or written.
  int result = tcflush(fd, TCIOFLUSH);
  if (result)
  {
    perror("tcflush failed");  // just a warning, not a fatal error
  }

  // Get the current configuration of the serial port.
  struct termios options;
  result = tcgetattr(fd, &options);
  if (result)
  {
    perror("tcgetattr failed");
    close(fd);
    return -1;
  }

  // Turn off any options that might interfere with our ability to send and
  // receive raw binary bytes.
  options.c_iflag &= ~(INLCR | IGNCR | ICRNL | IXON | IXOFF);
  options.c_oflag &= ~(ONLCR | OCRNL);
  options.c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);

  // Set up timeouts: Calls to read() will return as soon as there is
  // at least one byte available or when 100 ms has passed.
  options.c_cc[VTIME] = 1;
  options.c_cc[VMIN] = 0;

  // This code only supports certain standard baud rates. Supporting
  // non-standard baud rates should be possible but takes more work.
  switch (baud_rate)
  {
  case 4800:   cfsetospeed(&options, B4800);   break;
  case 9600:   cfsetospeed(&options, B9600);   break;
  case 19200:  cfsetospeed(&options, B19200);  break;
  case 38400:  cfsetospeed(&options, B38400);  break;
  case 115200: cfsetospeed(&options, B115200); break;
  default:
    fprintf(stderr, "warning: baud rate %u is not supported, using 9600.\n",
```

```
      baud_rate);
    cfsetospeed(&options, B9600);
    break;
  }
  cfsetispeed(&options, cfgetospeed(&options));

  result = tcsetattr(fd, TCSANOW, &options);
  if (result)
  {
    perror("tcsetattr failed");
    close(fd);
    return -1;
  }

  return fd;
}

// Writes bytes to the serial port, returning 0 on success and -1 on failure.
int write_port(int fd, const uint8_t * buffer, size_t size)
{
  ssize_t result = write(fd, buffer, size);
  if (result != (ssize_t)size)
  {
    perror("failed to write to port");
    return -1;
  }
  return 0;
}

// Reads bytes from the serial port.
// Returns after all the desired bytes have been read, or if there is a
// timeout or other error.
// Returns the number of bytes successfully read into the buffer, or -1 if
// there was an error reading.
ssize_t read_port(int fd, uint8_t * buffer, size_t size)
{
  size_t received = 0;
  while (received < size)
  {
    ssize_t r = read(fd, buffer + received, size - received);
    if (r < 0)
    {
      perror("failed to read from port");
      return -1;
    }
    if (r == 0)
    {
      // Timeout
      break;
    }
    received += r;
  }
  return received;
}

// Reads a variable from the SMC.
// Returns 0 on success or -1 on failure.
int smc_get_variable(int fd, uint8_t variable_id, uint16_t * value)
{
  uint8_t command[] = { 0xA1, variable_id };
  int result = write_port(fd, command, sizeof(command));
  if (result) { return -1; }
```

```
  uint8_t response[2];
  ssize_t received = read_port(fd, response, sizeof(response));
  if (received < 0) { return -1; }
  if (received != 2)
  {
    fprintf(stderr, "read timeout: expected 2 bytes, got %zu\n", received);
    return -1;
  }
  *value = response[0] + 256 * response[1];
  return 0;
}

// Gets the target speed (-3200 to 3200).
// Returns 0 on success, -1 on failure.
int smc_get_target_speed(int fd, int16_t * value)
{
  return smc_get_variable(fd, 20, (uint16_t *)value);
}

// Gets a number where each bit represents a different error, and the
// bit is 1 if the error is currently active.
// See the user's guide for definitions of the different error bits.
// Returns 0 on success, -1 on failure.
int smc_get_error_status(int fd, uint16_t * value)
{
  return smc_get_variable(fd, 0, value);
}

// Sends the Exit Safe Start command, which is required to drive the motor.
// Returns 0 on success, -1 on failure.
int smc_exit_safe_start(int fd)
{
  const uint8_t command = 0x83;
  return write_port(fd, &command, 1);
}

// Sets the SMC's target speed (-3200 to 3200).
// Returns 0 on success, -1 on failure.
int smc_set_target_speed(int fd, int speed)
{
  uint8_t command[3];

  if (speed < 0)
  {
    command[0] = 0x86; // Motor Reverse
    speed = -speed;
  }
  else
  {
    command[0] = 0x85; // Motor Forward
  }
  command[1] = speed & 0x1F;
  command[2] = speed >> 5 & 0x7F;

  return write_port(fd, command, sizeof(command));
}

int main()
{
  // Choose the serial port name.
  // Linux USB example:  "/dev/ttyACM0"  (see also: /dev/serial/by-id)
  // macOS USB example:  "/dev/cu.usbmodem001234562"
```

```
187     // Cygwin example:     "/dev/ttyS7"
188     const char * device = "/dev/ttyACM0";
189
190     // Choose the baud rate (bits per second).  This does not matter if you are
191     // connecting to the SMC over USB.  If you are connecting via the TX and RX
192     // lines, this should match the baud rate in the SMC G2's serial settings.
193     uint32_t baud_rate = 9600;
194
195     int fd = open_serial_port(device, baud_rate);
196     if (fd < 0) { return 1; }
197
198     int result = smc_exit_safe_start(fd);
199     if (result) { return 1; }
200
201     uint16_t error_status;
202     result = smc_get_error_status(fd, &error_status);
203     if (result) { return 1; }
204     printf("Error status: 0x%04x\n", error_status);
205
206     int16_t target_speed;
207     result = smc_get_target_speed(fd, &target_speed);
208     if (result) { return 1; }
209     printf("Target speed is %d.\n", target_speed);
210
211     int16_t new_speed = (target_speed <= 0) ? 3200 : -3200;
212     printf("Setting target speed to %d.\n", new_speed);
213     result = smc_set_target_speed(fd, new_speed);
214     if (result) { return 1; }
215
216     close(fd);
217     return 0;
218 }
```

8.7. Example serial code for Windows in C

The example C code below uses the Windows API to communicate with the Simple Motor Controller G2 via serial. It demonstrates how to use the USB virtual serial port or the TTL serial port to get the error status from the controller, read a variable, and set the target speed.

For this example to work, the Simple Motor Controller G2's input mode must be **Serial/USB**, the serial mode must be **Binary**, and the CRC must be disabled. These are the default settings that the controller is shipped with. If you are using TTL serial, you should set the controller to use a fixed baud rate of 9600, or change the baud rate in the code below to match whatever fixed baud rate you choose.

```c
// Uses Windows API serial functions to send and receive data from a
// Simple Motor Controller G2.
// NOTE: The Simple Motor Controller's input mode must be set to Serial/USB.
// NOTE: You must change the 'const char * device' line below.

#include <stdio.h>
#include <stdint.h>
#include <windows.h>

void print_error(const char * context)
{
  DWORD error_code = GetLastError();
  char buffer[256];
  DWORD size = FormatMessageA(
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_MAX_WIDTH_MASK,
    NULL, error_code, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
    buffer, sizeof(buffer), NULL);
  if (size == 0) { buffer[0] = 0; }
  fprintf(stderr, "%s: %s\n", context, buffer);
}

// Opens the specified serial port, configures its timeouts, and sets its
// baud rate.  Returns a handle on success, or INVALID_HANDLE_VALUE on failure.
HANDLE open_serial_port(const char * device, uint32_t baud_rate)
{
  HANDLE port = CreateFileA(device, GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  if (port == INVALID_HANDLE_VALUE)
  {
    print_error(device);
    return INVALID_HANDLE_VALUE;
  }

  // Flush away any bytes previously read or written.
  BOOL success = FlushFileBuffers(port);
  if (!success)
  {
    print_error("Failed to flush serial port");
    CloseHandle(port);
    return INVALID_HANDLE_VALUE;
  }

  // Configure read and write operations to time out after 100 ms.
  COMMTIMEOUTS timeouts = { 0 };
  timeouts.ReadIntervalTimeout = 0;
  timeouts.ReadTotalTimeoutConstant = 100;
  timeouts.ReadTotalTimeoutMultiplier = 0;
  timeouts.WriteTotalTimeoutConstant = 100;
  timeouts.WriteTotalTimeoutMultiplier = 0;

  success = SetCommTimeouts(port, &timeouts);
  if (!success)
  {
    print_error("Failed to set serial timeouts");
    CloseHandle(port);
    return INVALID_HANDLE_VALUE;
  }

  DCB state;
  state.DCBlength = sizeof(DCB);
  success = GetCommState(port, &state);
  if (!success)
```

```
  {
    print_error("Failed to get serial settings");
    CloseHandle(port);
    return INVALID_HANDLE_VALUE;
  }

  state.BaudRate = baud_rate;

  success = SetCommState(port, &state);
  if (!success)
  {
    print_error("Failed to set serial settings");
    CloseHandle(port);
    return INVALID_HANDLE_VALUE;
  }

  return port;
}

// Writes bytes to the serial port, returning 0 on success and -1 on failure.
int write_port(HANDLE port, const uint8_t * buffer, size_t size)
{
  DWORD written;
  BOOL success = WriteFile(port, buffer, size, &written, NULL);
  if (!success)
  {
    print_error("Failed to write to port");
    return -1;
  }
  if (written != size)
  {
    print_error("Failed to write all bytes to port");
    return -1;
  }
  return 0;
}

// Reads bytes from the serial port.
// Returns after all the desired bytes have been read, or if there is a
// timeout or other error.
// Returns the number of bytes successfully read into the buffer, or -1 if
// there was an error reading.
SSIZE_T read_port(HANDLE port, uint8_t * buffer, size_t size)
{
  DWORD received;
  BOOL success = ReadFile(port, buffer, size, &received, NULL);
  if (!success)
  {
    print_error("Failed to read from port");
    return -1;
  }
  return received;
}

// Reads a variable from the SMC.
// Returns 0 on success or -1 on failure.
int smc_get_variable(HANDLE port, uint8_t variable_id, uint16_t * value)
{
  uint8_t command[] = { 0xA1, variable_id };
  int result = write_port(port, command, sizeof(command));
  if (result) { return -1; }
  uint8_t response[2];
```

```
  SSIZE_T received = read_port(port, response, sizeof(response));
  if (received < 0) { return -1; }
  if (received != 2)
  {
    fprintf(stderr, "read timeout: expected 2 bytes, got %ld\n", received);
    return -1;
  }
  *value = response[0] + 256 * response[1];
  return 0;
}

// Gets the target speed (-3200 to 3200).
// Returns 0 on success, -1 on failure.
int smc_get_target_speed(HANDLE port, int16_t * value)
{
  return smc_get_variable(port, 20, (uint16_t *)value);
}

// Gets a number where each bit represents a different error, and the
// bit is 1 if the error is currently active.
// See the user's guide for definitions of the different error bits.
// Returns 0 on success, -1 on failure.
int smc_get_error_status(HANDLE port, uint16_t * value)
{
  return smc_get_variable(port, 0, value);
}

// Sends the Exit Safe Start command, which is required to drive the motor.
// Returns 0 on success, -1 on failure.
int smc_exit_safe_start(HANDLE port)
{
  const uint8_t command = 0x83;
  return write_port(port, &command, 1);
}

// Sets the SMC's target speed (-3200 to 3200).
// Returns 0 on success, -1 on failure.
int smc_set_target_speed(HANDLE port, int speed)
{
  uint8_t command[3];

  if (speed < 0)
  {
    command[0] = 0x86; // Motor Reverse
    speed = -speed;
  }
  else
  {
    command[0] = 0x85; // Motor Forward
  }
  command[1] = speed & 0x1F;
  command[2] = speed >> 5 & 0x7F;

  return write_port(port, command, sizeof(command));
}

int main()
{
  // Choose the serial port name.
  const char * device = "\\\\.\\COM17";

  // Choose the baud rate (bits per second).  This does not matter if you are
```

```
187     // connecting to the SMC over USB.  If you are connecting via the TX and RX
188     // lines, this should match the baud rate in the SMC G2's serial settings.
189     uint32_t baud_rate = 9600;
190
191     HANDLE port = open_serial_port(device, baud_rate);
192     if (port == INVALID_HANDLE_VALUE) { return 1; }
193
194     int result = smc_exit_safe_start(port);
195     if (result) { return 1; }
196
197     uint16_t error_status;
198     result = smc_get_error_status(port, &error_status);
199     if (result) { return 1; }
200     printf("Error status: 0x%04x\n", error_status);
201
202     int16_t target_speed;
203     result = smc_get_target_speed(port, &target_speed);
204     if (result) { return 1; }
205     printf("Target speed is %d.\n", target_speed);
206
207     int16_t new_speed = (target_speed <= 0) ? 3200 : -3200;
208     printf("Setting target speed to %d.\n", new_speed);
209     result = smc_set_target_speed(port, new_speed);
210     if (result) { return 1; }
211
212     CloseHandle(port);
213     return 0;
214 }
```

8.8. Example serial code in Python

The example Python code below uses the **pySerial** **[http://pyserial.readthedocs.io]** library to communicate with the Simple Motor Controller G2 via serial. It demonstrates how to read variables from the controller and set its target speed.

For this example to work, the Simple Motor Controller G2's input mode must be **Serial/USB**, the serial mode must be **Binary**, and the CRC must be disabled. These are the default settings that the controller is shipped with. If you are using TTL serial, you should set the controller to use a fixed baud rate of 9600, or change the baud rate in the code below to match whatever fixed baud rate you choose.

If you run the code and get the error "ImportError: No module named serial" or "ModuleNotFoundError: No module named 'serial'", it means that the pySerial library is not installed, and you should follow the instructions in the **pySerial documentation** **[http://pyserial.readthedocs.io]** to install it.

```python
# Uses the pySerial library to send and receive data from a
# Simple Motor Controller G2.
#
# NOTE: The Simple Motor Controller's input mode must be "Serial/USB".
# NOTE: You might need to change the "port_name =" line below to specify the
#   right serial port.

import serial

class SmcG2Serial(object):
  def __init__(self, port, device_number=None):
    self.port = port
    self.device_number = device_number

  def send_command(self, cmd, *data_bytes):
    if self.device_number == None:
      header = [cmd]  # Compact protocol
    else:
      header = [0xAA, device_number, cmd & 0x7F]  # Pololu protocol
    self.port.write(bytes(header + list(data_bytes)))

  # Sends the Exit Safe Start command, which is required to drive the motor.
  def exit_safe_start(self):
    self.send_command(0x83)

  # Sets the SMC's target speed (-3200 to 3200).
  def set_target_speed(self, speed):
    cmd = 0x85  # Motor forward
    if speed < 0:
      cmd = 0x86  # Motor reverse
      speed = -speed
    self.send_command(cmd, speed & 0x1F, speed >> 5 & 0x7F)

  # Gets the specified variable as an unsigned value.
  def get_variable(self, id):
    self.send_command(0xA1, id)
    result = self.port.read(2)
    if len(result) != 2:
      raise RuntimeError("Expected to read 2 bytes, got {}."
        .format(len(result)))
    b = bytearray(result)
    return b[0] + 256 * b[1]

  # Gets the specified variable as a signed value.
  def get_variable_signed(self, id):
    value = self.get_variable(id)
    if value >= 0x8000:
      value -= 0x10000
    return value

  # Gets the target speed (-3200 to 3200).
  def get_target_speed(self):
    return self.get_variable_signed(20)

  # Gets a number where each bit represents a different error, and the
  # bit is 1 if the error is currently active.
  # See the user's guide for definitions of the different error bits.
  def get_error_status(self):
    return self.get_variable(0)

# Choose the serial port name.
# Linux USB example:  "/dev/ttyACM0"  (see also: /dev/serial/by-id)
```

```
# macOS USB example:  "/dev/cu.usbmodem001234562"
# Windows example:    "COM6"
port_name = "/dev/ttyACM0"

# Choose the baud rate (bits per second).  This does not matter if you are
# connecting to the SMC over USB.  If you are connecting via the TX and RX
# lines, this should match the baud rate in the SMC's serial settings.
baud_rate = 9600

# Change this to a number between 0 and 127 that matches the device number of
# your SMC if there are multiple serial devices on the line and you want to
# use the Pololu Protocol.
device_number = None

port = serial.Serial(port_name, baud_rate, timeout=0.1, write_timeout=0.1)

smc = SmcG2Serial(port, device_number)

smc.exit_safe_start()

error_status = smc.get_error_status()
print("Error status: 0x{:04X}".format(error_status))

target_speed = smc.get_target_speed()
print("Target speed is {}.".format(target_speed))

new_speed = 3200 if target_speed <= 0 else -3200
print("Setting target speed to {}.\n".format(new_speed));
smc.set_target_speed(new_speed)
```

8.9. Example serial code for Linux or macOS in Bash

The Bash shell script below works on Linux and macOS. It demonstrates how to control a Simple Motor Controller G2 using its USB virtual serial port.

For this script to work, the Simple Motor Controller G2's input mode must be **Serial/USB**, the serial mode must be **Binary**, and CRC must be disabled. These are the default settings that the controller is shipped with. The controller should be connected to the computer via USB.

```
#!/bin/bash
# Sets the speed of a Simple Motor Controller via its virtual serial port.
# Usage: smc-set-speed.sh DEVICE SPEED
# Linux example: bash smc-set-speed.sh /dev/ttyACM0 3200
# macOS example: bash smc-set-speed.sh /dev/cu.usbmodemfa121 3200
# Cygwin example: bash smc-set-speed.sh '/dev/ttyS16' 3200
# DEVICE is the name of the virtual COM port device.
# SPEED is a number between -3200 and 3200
DEVICE=$1
SPEED=$2

byte() {
  printf "\\x$(printf "%x" $1)"
}

{
  byte 0x83  # exit safe-start
  if [ $SPEED -lt 0 ]; then
    byte 0x86  # motor reverse
    SPEED=$((-$SPEED))
  else
    byte 0x85  # motor forward
```

```
  fi
  byte $((SPEED & 0x1F))
  byte $((SPEED >> 5 & 0x7F))
} > $DEVICE
```

8.10. Example I²C code for Arduino

This section shows how to use with the Simple Motor Controller G2's I²C interface from an **Arduino [https://www.pololu.com/category/125/arduino]**, **A-Star [https://www.pololu.com/category/149/a-star-programmable-controllers]**, or other Arduino-compatible controller.

To use this code, you must check the "Enable I2C" checkbox in the "Input settings" tab of the Simple Motor Control Center G2. You should make sure that "Enable CRC for commands" is not checked. You should make sure that the controller's "Device number" setting is 13, or else change the `smcDeviceNumber` constant in the program to match the device number.

This example assumes the following connections exist between the Arduino and the Simple Motor Controller G2:

- Arduino SCL to Simple Motor Controller **SCL**
- Arduino SDA to Simple Motor Controller **SDA/RX**
- Arduino **GND** to Simple Motor Controller **GND**

If you are not sure which pins are SCL and SDA on your Arduino, refer to the documentation of your board and to the **Wire library documentation [https://www.arduino.cc/en/Reference/Wire]**.

See **Section 4.4** for more information on connecting an I²C device to the Simple Motor Controller G2.

This program demonstrates how to send I²C commands to set the motor speed and read variables. For information about the commands used by this sample code, refer to **Section 6.2.1**. Note that the Simple Motor Controller G2 must be powered when this Arduino sketch starts running.

```ino
#include <Wire.h>

const uint8_t smcDeviceNumber = 13;

// Required to allow motors to move.
// Must be called when controller restarts and after any error.
void exitSafeStart()
{
  Wire.beginTransmission(smcDeviceNumber);
  Wire.write(0x83);  // Exit safe start
  Wire.endTransmission();
}

void setMotorSpeed(int16_t speed)
{
  uint8_t cmd = 0x85;  // Motor forward
  if (speed < 0)
  {
    cmd = 0x86;  // Motor reverse
    speed = -speed;
  }
  Wire.beginTransmission(smcDeviceNumber);
  Wire.write(cmd);
  Wire.write(speed & 0x1F);
  Wire.write(speed >> 5 & 0x7F);
  Wire.endTransmission();
}

uint16_t readUpTime()
{
  Wire.beginTransmission(smcDeviceNumber);
  Wire.write(0xA1);  // Command: Get variable
  Wire.write(28);    // Variable ID: Up time (low)
  Wire.endTransmission();
  Wire.requestFrom(smcDeviceNumber, (uint8_t)2);
  uint16_t upTime = Wire.read();
  upTime |= Wire.read() << 8;
  return upTime;
}

void setup()
{
  Wire.begin();
  exitSafeStart();
}

void loop()
{
  // Read the up time from the controller and send it to
  // the serial monitor.
  uint16_t upTime = readUpTime();
  Serial.print(F("Up time: "));
  Serial.println(upTime);

  setMotorSpeed(3200);   // full-speed forward
  delay(1000);
  setMotorSpeed(-3200);  // full-speed reverse
  delay(1000);
}
```

8.11. Example I²C code for Linux in C

The example C code below uses the I²C API provided by the Linux kernel to send and receive data from a Simple Motor Controller G2. This code only works on Linux.

If you are using a Raspberry Pi, please note that the Raspberry Pi's hardware I²C module has a **bug [https://github.com/raspberrypi/linux/issues/254]** that might cause this code to be unreliable. As a workaround, we recommend disabling hardware I²C (if you previously enabled it) and enabling the i2c-gpio overlay instead. To enable the overlay, add the following line to `/boot/config.txt` and reboot:

```
dtoverlay=i2c-gpio,i2c_gpio_sda=2,i2c_gpio_scl=3
```

The **Raspberry Pi overlay documentation [https://github.com/raspberrypi/firmware/blob/master/boot/overlays/README]** has information about the parameters on that line. Connect the Simple Motor Controller's SDA line to GPIO2 and connect the Simple Motor Controller's SCL line to GPIO3. The i2c-gpio overlay creates a new I²C bus device named `/dev/i2c-3`, and the code below uses that device. To give your user permission to access I²C busses without being root, you might have to add yourself to the `i2c` group by running `sudo usermod -a -G i2c $(whoami)` and restarting.

```c
// Uses the Linux I2C API to send and receive data from an SMC G2.
// NOTE: The SMC's input mode must be "Serial / I2C / USB".
// NOTE: For reliable operation on a Raspberry Pi, enable the i2c-gpio
//   overlay and use the I2C device it provides (usually /dev/i2c-3).
// NOTE: You might need to change the 'const char * device' line below
//   to specify the correct I2C device.
// NOTE: You might need to change the `const uint8_t address' line below
// to match the device number of your Simple Motor Controller.

#include <fcntl.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
#include <stdint.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>

// Opens the specified I2C device.  Returns a non-negative file descriptor
// on success, or -1 on failure.
int open_i2c_device(const char * device)
{
  int fd = open(device, O_RDWR);
  if (fd == -1)
  {
    perror(device);
    return -1;
  }
  return fd;
}

// Reads a variable from the SMC.
// Returns 0 on success or -1 on failure.
int smc_get_variable(int fd, uint8_t address, uint8_t variable_id,
  uint16_t * value)
{
  uint8_t command[] = { 0xA1, variable_id };
  uint8_t response[2];
  struct i2c_msg messages[] = {
    { address, 0, sizeof(command), command },
    { address, I2C_M_RD, sizeof(response), response },
  };
  struct i2c_rdwr_ioctl_data ioctl_data = { messages, 2 };
  int result = ioctl(fd, I2C_RDWR, &ioctl_data);
  if (result != 2)
  {
    perror("failed to get variable");
    return -1;
  }
  *value = response[0] + 256 * response[1];
  return 0;
}

// Gets the target speed (-3200 to 3200).
// Returns 0 on success, -1 on failure.
int smc_get_target_speed(int fd, uint8_t address, int16_t * value)
{
  return smc_get_variable(fd, address, 20, (uint16_t *)value);
}

// Gets a number where each bit represents a different error, and the
// bit is 1 if the error is currently active.
// See the user's guide for definitions of the different error bits.
```

```
63  // Returns 0 on success, -1 on failure.
64  int smc_get_error_status(int fd, uint8_t address, uint16_t * value)
65  {
66    return smc_get_variable(fd, address, 0, value);
67  }
68
69  // Sends the Exit Safe Start command, which is required to drive the motor.
70  // Returns 0 on success, -1 on failure.
71  int smc_exit_safe_start(int fd, uint8_t address)
72  {
73    uint8_t command = 0x83;
74    struct i2c_msg message = { address, 0, 1, &command };
75    struct i2c_rdwr_ioctl_data ioctl_data = { &message, 1 };
76    int result = ioctl(fd, I2C_RDWR, &ioctl_data);
77    if (result != 1)
78    {
79      perror("failed to exit safe start");
80      return -1;
81    }
82    return 0;
83  }
84
85  // Sets the SMC's target speed (-3200 to 3200).
86  // Returns 0 on success, -1 on failure.
87  int smc_set_target_speed(int fd, uint8_t address, int16_t speed)
88  {
89    uint8_t command[3];
90
91    if (speed < 0)
92    {
93      command[0] = 0x86; // Motor Reverse
94      speed = -speed;
95    }
96    else
97    {
98      command[0] = 0x85; // Motor Forward
99    }
100   command[1] = speed & 0x1F;
101   command[2] = speed >> 5 & 0x7F;
102
103   struct i2c_msg message = { address, 0, sizeof(command), command };
104   struct i2c_rdwr_ioctl_data ioctl_data = { &message, 1 };
105   int result = ioctl(fd, I2C_RDWR, &ioctl_data);
106   if (result != 1)
107   {
108     perror("failed to set speed");
109     return -1;
110   }
111   return 0;
112 }
113
114 int main()
115 {
116   // Choose the I2C device.
117   const char * device = "/dev/i2c-3";
118
119   // Set the I2C address of the SMC (the device number).
120   const uint8_t address = 13;
121
122   int fd = open_i2c_device(device);
123   if (fd < 0) { return 1; }
124
```

```
125     int result = smc_exit_safe_start(fd, address);
126     if (result) { return 1; }
127
128     uint16_t error_status;
129     result = smc_get_error_status(fd, address, &error_status);
130     if (result) { return 1; }
131     printf("Error status: 0x%04x\n", error_status);
132
133     int16_t target_speed;
134     result = smc_get_target_speed(fd, address, &target_speed);
135     if (result) { return 1; }
136     printf("Target speed is %d.\n", target_speed);
137
138     int16_t new_speed = (target_speed <= 0) ? 3200 : -3200;
139     printf("Setting target speed to %d.\n", new_speed);
140     result = smc_set_target_speed(fd, address, new_speed);
141     if (result) { return 1; }
142
143     close(fd);
144     return 0;
145 }
```

8.12. Example I²C code for Linux in Python

The example code below uses a Python library named smbus2 to communicate with a Simple Motor Controller G2 via I²C. This example works on Linux with either Python 2 or Python 3.

If you are using a Raspberry Pi, please see the notes about setting up I²C for the Raspberry Pi in **Section 8.11**.

To install the smbus2 library, you will need to run either `pip install smbus2` or `pip3 install smbus2` depending on what version of Python you want to use and what Linux distribution you are using. On Raspbian, `pip` is for Python 2 and `pip3` is for Python 3.

```python
# Uses the smbus2 library to send and receive data from a
# Simple Motor Controller G2.
# Works on Linux with either Python 2 or Python 3.
#
# NOTE: The SMC's input mode must be "Serial/USB".
# NOTE: You might nee to change the 'SMBus(3)' line below to specify the
#   correct I2C bus device.
# NOTE: You might need to change the 'address = 13' line below to match
#   the device number of your Simple Motor Controller.

from smbus2 import SMBus, i2c_msg

class SmcG2I2C(object):
  def __init__(self, bus, address):
    self.bus = bus
    self.address = address

  # Sends the Exit Safe Start command, which is required to drive the motor.
  def exit_safe_start(self):
    write = i2c_msg.write(self.address, [0x83])
    self.bus.i2c_rdwr(write)

  # Sets the SMC's target speed (-3200 to 3200).
  def set_target_speed(self, speed):
    cmd = 0x85  # Motor forward
    if speed < 0:
      cmd = 0x86  # Motor reverse
      speed = -speed
    buffer = [cmd, speed & 0x1F, speed >> 5 & 0x7F]
    write = i2c_msg.write(self.address, buffer)
    self.bus.i2c_rdwr(write)

  # Gets the specified variable as an unsigned value.
  def get_variable(self, id):
    write = i2c_msg.write(self.address, [0xA1, id])
    read = i2c_msg.read(self.address, 2)
    self.bus.i2c_rdwr(write, read)
    b = list(read)
    return b[0] + 256 * b[1]

  # Gets the specified variable as a signed value.
  def get_variable_signed(self, id):
    value = self.get_variable(id)
    if value >= 0x8000:
      value -= 0x10000
    return value

  # Gets the target speed (-3200 to 3200).
  def get_target_speed(self):
    return self.get_variable_signed(20)

  # Gets a number where each bit represents a different error, and the
  # bit is 1 if the error is currently active.
  # See the user's guide for definitions of the different error bits.
  def get_error_status(self):
    return self.get_variable(0)

# Open a handle to "/dev/i2c-3", representing the I2C bus.
bus = SMBus(3)

# Select the I2C address of the Simple Motor Controller (the device number).
address = 13
```

```
63
64  smc = SmcG2I2C(bus, address)
65
66  smc.exit_safe_start()
67
68  error_status = smc.get_error_status()
69  print("Error status: 0x{:04X}".format(error_status))
70
71  target_speed = smc.get_target_speed()
72  print("Target speed is {}.".format(target_speed))
73
74  new_speed = 3200 if target_speed <= 0 else -3200
75  print("Setting target speed to {}.\n".format(new_speed));
76  smc.set_target_speed(new_speed)
```

8.13. Example CRC computation in C

Simple Example

The following example program shows how to compute a CRC byte in the C language. The outer loop processes each byte, and the inner loop processes each bit of those bytes. In the example main() routine, this is applied to generate the CRC byte in the message 0x83, 0x01, that was used in **Section 6.5**. The `getCRC()` function will work without modification in both Arduino and Orangutan programs.

```
1   const unsigned char CRC7_POLY = 0x91;
2
3   unsigned char getCRC(unsigned char message[], unsigned char length)
4   {
5     unsigned char i, j, crc = 0;
6
7     for (i = 0; i < length; i++)
8     {
9       crc ^= message[i];
10      for (j = 0; j < 8; j++)
11      {
12        if (crc & 1)
13          crc ^= CRC7_POLY;
14        crc >>= 1;
15      }
16    }
17    return crc;
18  }
19
20  int main()
21  {
22    // create a message array that has one extra byte to hold the CRC:
23    unsigned char message[3] = {0x83, 0x01, 0x00};
24    message[2] = getCRC(message, 2);
25    // send this message to the Simple Motor Controller
26  }
```

Advanced Example

The following example program shows a more efficient way to compute a CRC in the C language. The increased efficiency is achieved by pre-computing the CRCs of all 256 possible bytes and storing them in a lookup table, which can be in RAM, flash, or EEPROM. These table values are then XORed

together based on the bytes of the message to get the final CRC. In the example main() routine, this is applied to generate the CRC byte in the message 0x83, 0x01, that was used in **Section 6.5**.

```
#include <stdio.h>

const unsigned char CRC7_POLY = 0x91;
unsigned char CRCTable[256];

unsigned char getCRCForByte(unsigned char val)
{
  unsigned char j;

  for (j = 0; j < 8; j++)
  {
    if (val & 1)
      val ^= CRC7_POLY;
    val >>= 1;
  }

  return val;
}

void buildCRCTable()
{
  int i;

  // fill an array with CRC values of all 256 possible bytes
  for (i = 0; i < 256; i++)
  {
    CRCTable[i] = getCRCForByte(i);
  }
}

unsigned char getCRC(unsigned char message[], unsigned char length)
{
  unsigned char i, crc = 0;

  for (i = 0; i < length; i++)
    crc = CRCTable[crc ^ message[i]];
  return crc;
}

int main()
{
  unsigned char message[3] = {0x83, 0x01, 0x00};
  int i, j;

  buildCRCTable();
  message[2] = getCRC(message, 2);

  for (i = 0; i < sizeof(message); i++)
  {
    for (j = 0; j < 8; j++)
      printf("%d", (message[i] >> j) % 2);
    printf(" ");
  }
  printf("\n");
}
```