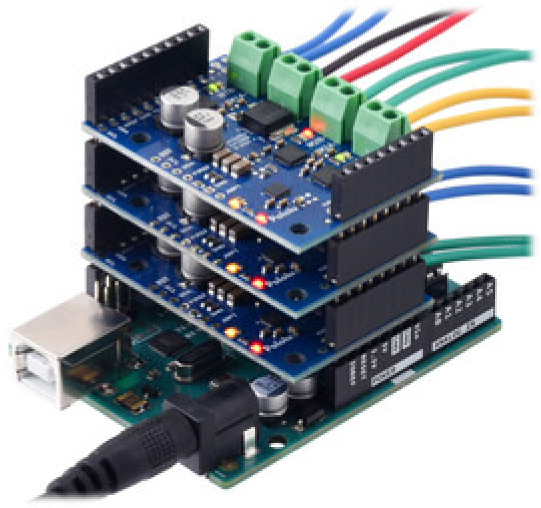
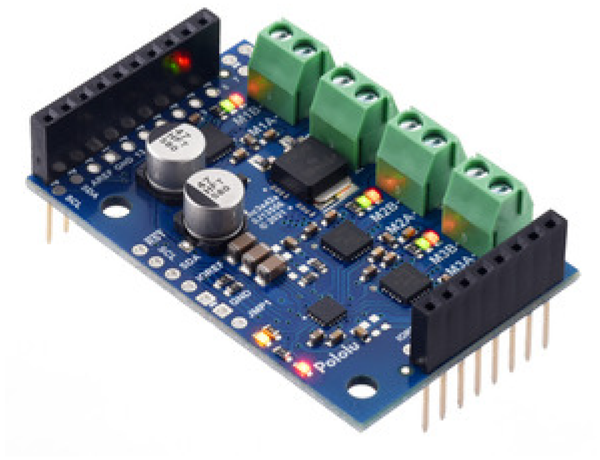
Pololu Motoron Motor Controller User’s Guide



1. Overview 2
1.1. Available versions 4
2. Contacting Pololu 11
3. Getting started 12
3.1. Choosing the power supply and motor 12
3.2. Connecting everything 13
3.3. Enabling I²C on the Raspberry Pi 16
3.4. Setting I²C addresses with a Raspberry Pi 17
3.5. Setting I²C addresses with an Arduino 18
3.6. Writing code 20
4. Motoron M3S256 pinout 24
5. Motoron M3H256 pinout 26
6. LED feedback 28
7. I²C interface 30
8. Variable reference 32
9. Command reference 51
10. Cyclic redundancy check (CRC) 66
11. Reset pin 67

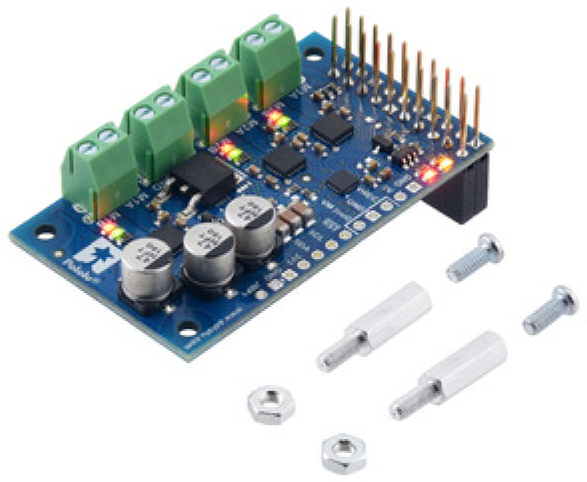
1. Overview

The Motoron motor controllers are a family of general-purpose modules designed to control brushed DC motors. The Motoron receives commands via I²C, so only two I/O lines are needed regardless of how many Motorons you connect. The Motoron modules are designed to plug into either a Raspberry Pi or an Arduino, and multiple Motoron controllers can be stacked on top of each other, allowing independent control of many motors.



Motoron M3S256 Triple Motor Controller Shield for Arduino (Connectors Soldered).

The **Motoron M3S256 Triple Motor Controller Shield for Arduino**, shown above, can control three motors and is designed to plug into an **Arduino** **[https://www.pololu.com/product/2191]** or compatible board, such as the **A-Star 32U4 Prime** **[https://www.pololu.com/category/165/a-star-32u4-prime]**. The Motoron M3S256 can operate from 4.5 V to 48 V and has reverse-voltage protection on motor power supply (down to −40 V). Its maximum continuous output current per motor is 2.0 A and it can provide a peak current of 6.4 A for less than 1 second.



Motoron M3H256 Triple Motor Controller for Raspberry Pi (Connectors Soldered).

The **Motoron M3H256 Triple Motor Controller for Raspberry Pi**, shown above, can control three motors and is designed to plug into a Raspberry Pi. The Motoron M3H256 can operate from 4.5 V to 48 V and has reverse-voltage protection on motor power supply (down to −40 V). Its maximum continuous output current per motor is 2.0 A and it can provide a peak current of 6.4 A for less than 1 second.

Features and specifications

- Logic voltage range: 2.8 V to 5.5 V
- Control interface: I²C
- I²C clock speed: up to 400 kHz
- Optional cyclic redundancy checking (CRC)
- Configurable motion parameters:
    - Max acceleration/deceleration forward/reverse
    - Starting speed forward/reverse
    - Direction change delay forward/reverse
- PWM frequency: eight options available from 1 kHz to 80 kHz
- Command timeout feature stops motors if the Arduino stops functioning
- Configurable automatic error response
- Motor power supply (VIN) voltage measurement
- Optional pins make it easy to power the Arduino or Raspberry Pi from reverse-protected

motor power, either directly or through an **external regulator** **[https://www.pololu.com/category/136/voltage-regulators]** (not included)

- Two status LEDs
- Motor direction indicator LEDs
- **Motoron Arduino library** **[https://github.com/pololu/motoron-arduino]** simplifies getting started with an Arduino or compatible controller
- **Motoron Python library** **[https://github.com/pololu/motoron-rpi]** simplifies getting started with a Raspberry Pi

1.1. Available versions

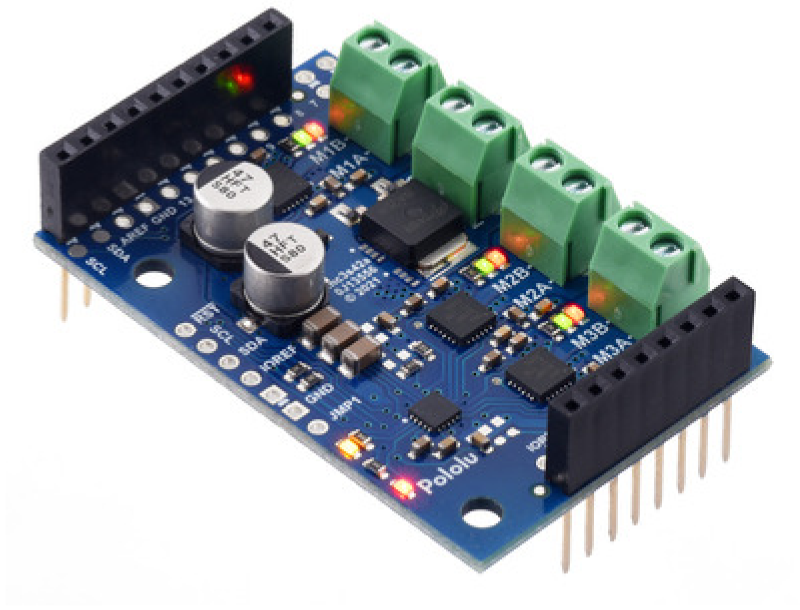
The Motoron family consists of two different controllers: the **Motoron M3S256** (an shield for Arduino) and the **Motoron M3H256** (for the Rasbperry Pi).

Multiple versions of each controller are available to provide different options for the through-hole connectors:

- **Motoron M3S256 with soldered stackable headers and terminal blocks** **[https://www.pololu.com/product/5030]**
- **Motoron M3S256 with headers and terminal blocks included but not soldered in** **[https://www.pololu.com/product/5031]**
- **Motoron M3S256 without any headers or terminal blocks included** **[https://www.pololu.com/product/5032]**
- **Motoron M3H256 with soldered stackable headers, soldered terminal blocks, and standoffs** **[https://www.pololu.com/product/5033]**
- **Motoron M3H256 with headers, terminal blocks, and standoffs included but not soldered in** **[https://www.pololu.com/product/5034]**
- **Motoron M3H256 without any standoffs, headers, or terminal blocks included** **[https://www.pololu.com/product/5035]**

See below for more information about these three options.

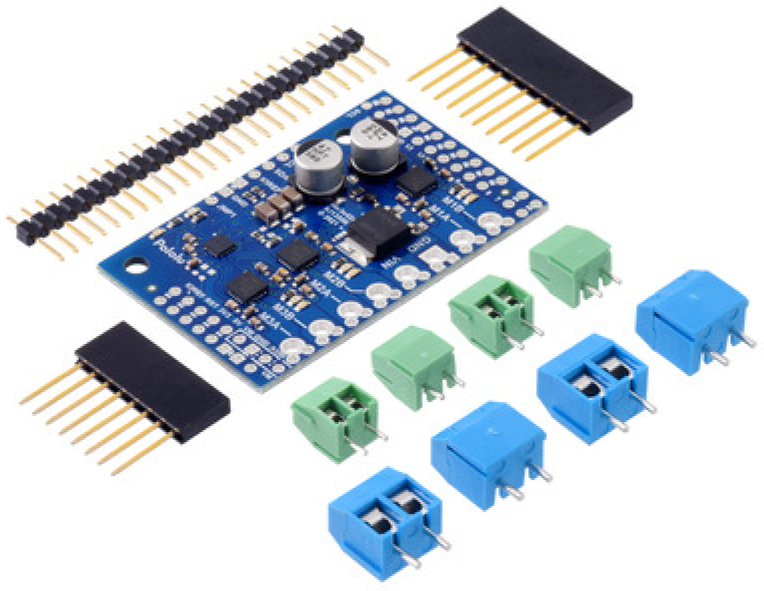
M3S256 with connectors soldered



Motoron M3S256 Triple Motor Controller Shield for Arduino (Connectors Soldered).

The **M3S256 version with connectors soldered [https://www.pololu.com/product/5030]** has stackable female headers and terminal blocks installed. This version allows you to use all the main features of the board without additional soldering, as the motor and power leads can be connected to the board via terminal blocks and the logic connections are made by simply plugging the shield into an Arduino.

M3S256 kit version



Motoron M3S256 Triple Motor Controller Shield Kit for Arduino.

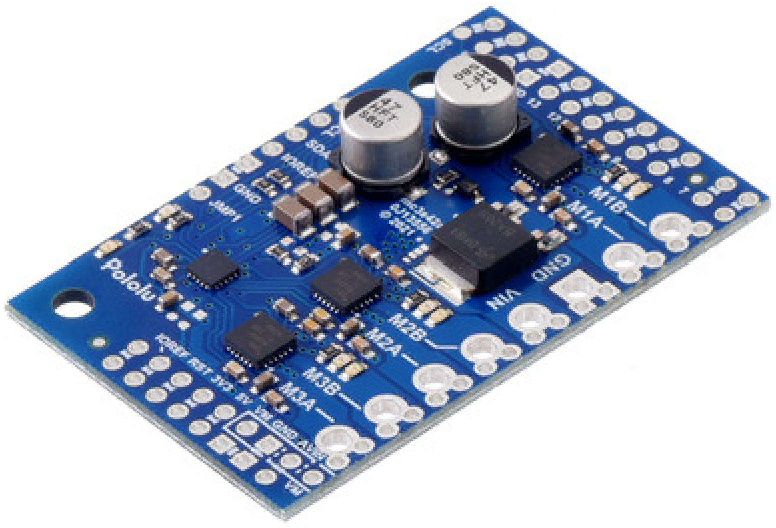
The **M3S256 kit version** **[https://www.pololu.com/product/5031]** comes with the following connectors included but not soldered, allowing for more installation options:

- One 1×10 stackable female header
- One 1×8 stackable female header
- Four **2-pin 3.5mm terminal blocks** **[https://www.pololu.com/product/2444]**
- Four **2-pin 5mm screw terminal blocks** **[https://www.pololu.com/product/2440]**
- One 1×25 **male header** **[https://www.pololu.com/product/965]**

There are more parts included than can be soldered to the board, providing different assembly options to suit a variety of applications. Additional **connector** **[https://www.pololu.com/category/19/connectors]** options are available separately, and wires can also be soldered directly to the board for more compact installations.

Note: For applications where the Motoron M3S256 will be used as a standalone board or at the **top** of a shield stack, it can be assembled with the included 0.1″ male header pins and larger (5.0mm-pitch) blue terminal blocks. For applications where the Motoron will be one of the intermediate members of a stack of shields, we recommend assembling it with the stackable headers and smaller (3.5mm-pitch) green terminal blocks (just like our **version with connectors already soldered** **[https://www.pololu.com/product/5030]**). The terminal blocks are intended to be soldered to the larger through-holes for the power and motor connections, and the blue ones should get locked together prior to installation (see our short **video on terminal block installation** **[https://www.youtube.com/watch?v=6pDyTLRZ2Eg]**).

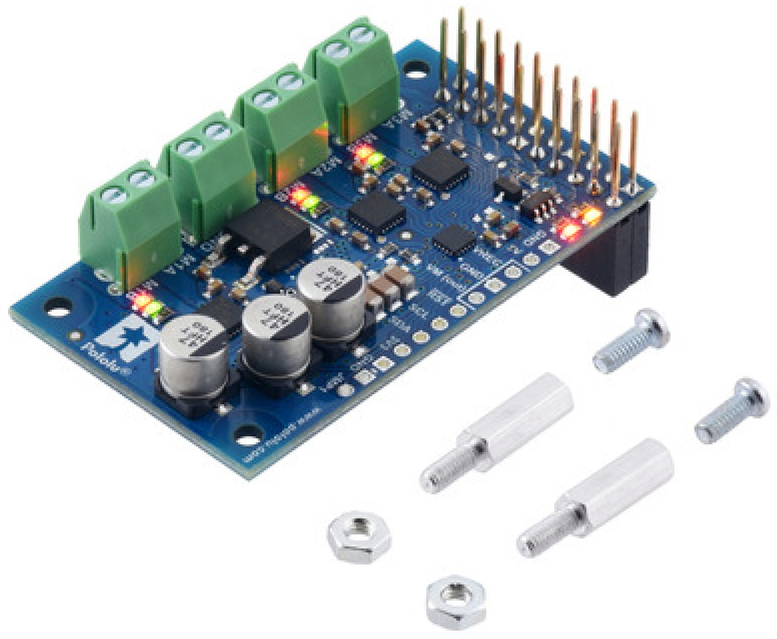
M3S256 with no connectors



Motoron M3S256 Triple Motor Controller Shield for Arduino (No Connectors).

The **"no connectors" version** **[https://www.pololu.com/product/5032]** is just the PCB assembled with all of the surface-mount components. This version is intended for those who want to solder wires directly to the board or use a custom set of **connectors** **[https://www.pololu.com/category/19/connectors]**.

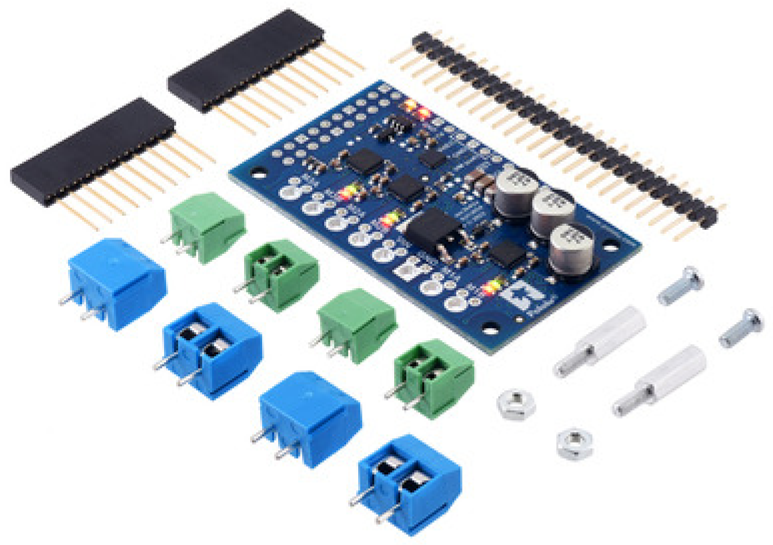
M3H256 with connectors soldered



Motoron M3H256 Triple Motor Controller for Raspberry Pi (Connectors Soldered).

The **M3H256 version with connectors soldered [https://www.pololu.com/product/5033]** has stackable female headers and terminal blocks installed, and it comes with two **standoffs [https://www.pololu.com/product/1952]**, two **screws [https://www.pololu.com/product/1968]**, and two **hex nuts [https://www.pololu.com/product/1967]**. This version allows you to use all the main features of the board without additional soldering, as the motor and power leads can be connected to the board via terminal blocks and the logic connections are made by simply plugging the shield into a Raspberry Pi.

M3H256 kit version



Motoron M3H256 Triple Motor Controller Kit for Raspberry Pi.

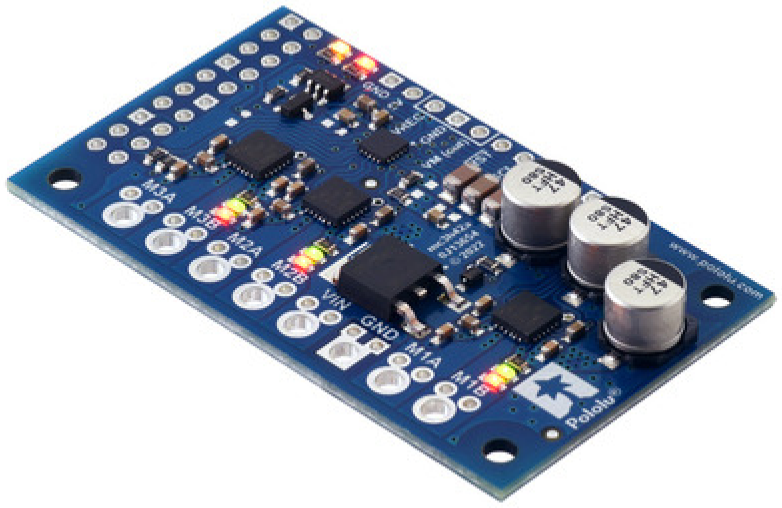
The **M3H256 kit version** **[https://www.pololu.com/product/5034]** comes with the following connectors included but not soldered, allowing for more installation options:

- Two 1×10 stackable female headers (or one 2×10 header)
- Four **2-pin 3.5mm terminal blocks** **[https://www.pololu.com/product/2444]**
- Four **2-pin 5mm screw terminal blocks** **[https://www.pololu.com/product/2440]**
- One 1×25 **male header** **[https://www.pololu.com/product/965]**
- Two **standoffs** **[https://www.pololu.com/product/1952]**
- Two **screws** **[https://www.pololu.com/product/1968]**
- Two **hex nuts** **[https://www.pololu.com/product/1967]**

There are more parts included than can be soldered to the board, providing different assembly options to suit a variety of applications. Additional **connector** **[https://www.pololu.com/category/19/connectors]** options are available separately, and wires can also be soldered directly to the board for more compact installations.

Note: For applications where the Motoron M3H256 will be used as a standalone board or at the **top** of a stack of boards, it can be assembled with the larger (5.0mm-pitch) blue terminal blocks. For applications where the Motoron will be one of the intermediate members of a stack of boards, we recommend assembling it with the smaller (3.5mm-pitch) green terminal blocks (just like our **version with connectors already soldered [https://www.pololu.com/product/5033]**). The terminal blocks are intended to be soldered to the larger through-holes for the power and motor connections, and the blue ones should get locked together prior to installation (see our short **video on terminal block installation [https://www.youtube.com/watch?v=6pDyTLRZ2Eg]**).

M3H256 with no connectors



Motoron M3H256 Triple Motor Controller for Raspberry Pi (No Connectors or Standoffs).

The **"no connectors" version [https://www.pololu.com/product/5035]** is just the PCB assembled with all of the surface-mount components. This version is intended for those who want to solder wires directly to the board or use a custom set of **connectors [https://www.pololu.com/category/19/connectors]**.

2. Contacting Pololu

We would be delighted to hear from you about any of your projects and about your experience with the Motoron. If you need technical support or have any feedback you would like to share, you can **contact us** **[https://www.pololu.com/contact]** directly or post on our **forum** **[https://forum.pololu.com/c/support/pololu-motor-controllers-drivers-and-motors]**. 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. Choosing the power supply and motor

The information in this section can help you select a **power supply [https://www.pololu.com/category/84/regulators-and-power-supplies]** and **motor [https://www.pololu.com/category/22/motors-and-gearboxes]** that will work with the Motoron.

The Motoron is designed to work with **brushed DC motors**. These motors have two terminals such that when a DC voltage is applied to the terminals, the motor spins.

When selecting the components of your system, you will need to consider the voltage and current ratings of each component:

- The **voltage range of your power supply** is the range of voltages you expect your power supply to produce while operating. There is usually some variation in the output voltage so you should treat it as a range instead of just a single number. In particular, keep in mind that a fully-charged battery might have a voltage that is significantly higher than its nominal voltage.
- The **current limit of a power supply** is how much current the power supply can provide. Note that the power supply will not force this amount of current through your system; the properties of the system and the voltage of the power supply determine how much current will flow, but there is a limit to how much current the power supply can provide.
- The **operating voltage range of a Motoron** is the range of voltages that can be supplied to the Motoron's VIN and GND pins, which power the motors. The operating voltage range of the Motoron M3S256 and M3H256 is **4.5 V to 48 V**. The Motoron requires a DC power supply.
- The **continuous current per motor of a Motoron** is the maximum amount of current that the Motoron can continuously provide to each motor. The continuous current per phase of the Motoron M3S256 and M3H256 is **2.0 A**.
- The **rated voltage of a DC motor** is the voltage at which the DC motor was designed to run. You can apply voltages to the motor that are higher or lower than its rated voltage, but higher voltages bring a risk of overheating the motor or reducing its lifetime.
- The **no-load current of a DC motor** is the current that the motor will draw if you apply the rated voltage to the motor while its output is not connected to anything.
- The **stall current of a DC motor** is the current that the motor will draw if you apply the rated voltage to the motor while forcing its output shaft to remain stationary.

There are guidelines you should be aware of when selecting the components of your system:

1. The voltage of your power supply should generally be greater than or equal to the rated

voltage of each DC motor. Otherwise, you will not get the full performance that the motor was designed for. If your power supply’s voltage is much higher than the rated voltage of a DC motor, you might account for that by using lower speeds for that motor in your commands to the Motoron.

2. The voltage of your power supply should be within the operating voltage range of the Motoron. Otherwise, the Motoron could malfunction or (in the case of high voltages) be damaged.
3. The typical current draw you expect for each motor should be less than the Motoron’s continuous current per motor. Each motor’s typical current draw will depend on your power supply voltage, the speeds you command the motor to move, and the current ratings of the motor. If a motor draws too much current for too long, there is a risk of triggering the Motoron’s overcurrent or overtemperature faults, which shut down the motor.
4. The current limit of the power supply should be higher than the typical total current draw for all the motors in your system. Furthermore, it is generally good for the current limit to be much higher than that so your system can smoothly handle the times where the motors are drawing *more* than the typical current, for example when they are accelerating or encountering extra resistance.

3.2. Connecting everything

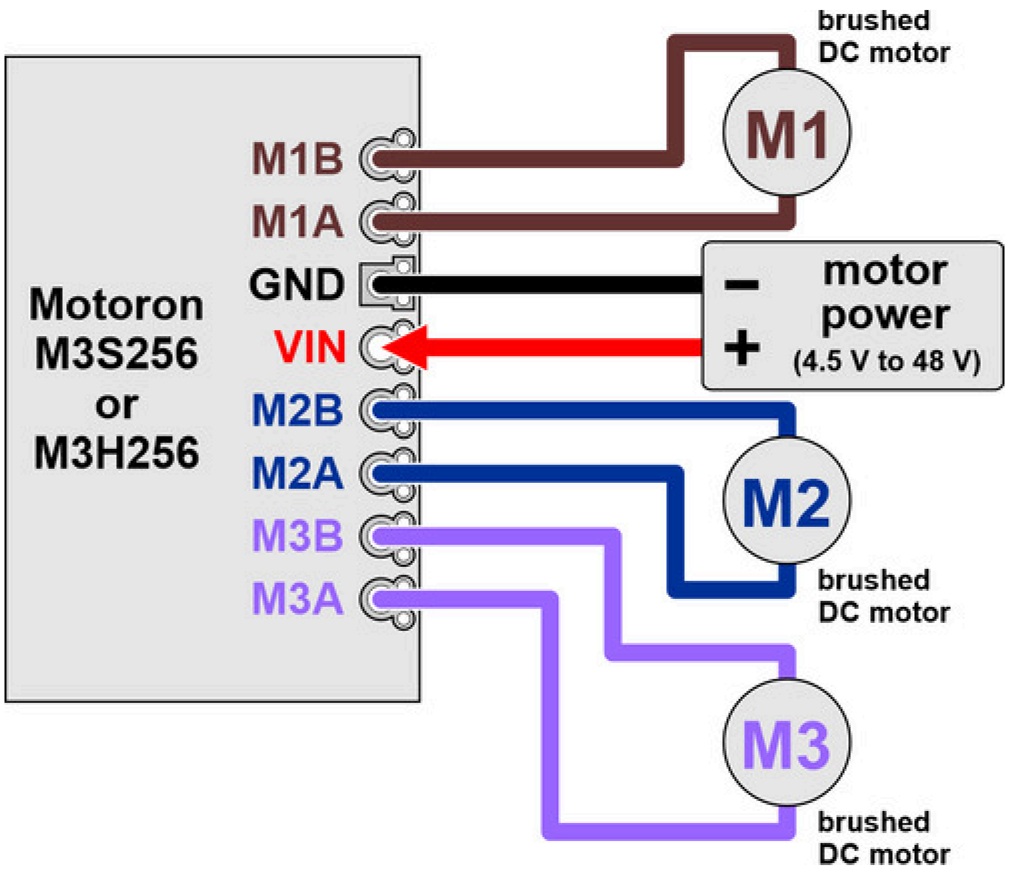
This section explains how to connect motor power, motors, and a microcontroller to a Motoron motor controller.

Connecting terminal blocks

We generally recommend using green **3.5mm-pitch terminal blocks [https://www.pololu.com/product/2444]** for the motor power and motor connections. If you have an assembled version of the Motoron, these terminal blocks come soldered to the board. Otherwise, you will need to solder them yourself. They should be soldered to the larger through holes for board power and motor outputs (GND, VIN, M1A, M1B, M2A, …).

Alternatively, if you are **not** going to stack multiple Motorons on top of each other, you can use blue **5mm-pitch terminal blocks [https://www.pololu.com/product/2440]**, which are included in the kit versions. The 5mm-pitch terminal blocks are tall enough that it is easy to accidentally cause a short between the motor outputs of two Motorons that are plugged into each other. If you decide to use the 5mm terminal blocks, we recommend using the tabs on the side of the terminal blocks to connect them together **before** soldering them to the Motoron (see our **video about how to install terminal blocks [https://www.youtube.com/watch?v=6pDyTLRZ2Eg]** for more information).

Connecting motor power and motors

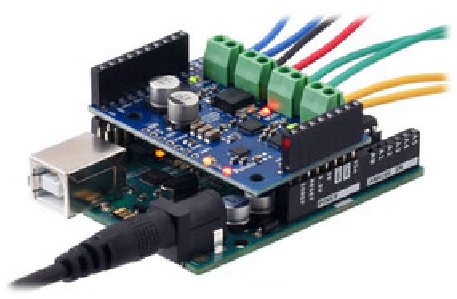


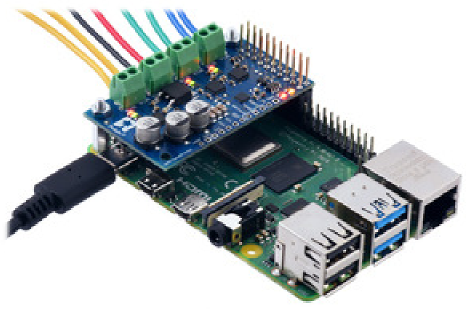
Motor power and motor connections for a Motoron M3S256 or M3H256 Triple Motor Controller.

The negative terminal of the motor power supply should be connected to the Motoron’s large GND pin or the smaller pins next to it. The positive terminal of the motor power supply should be connected to the Motoron’s large VIN pin or the smaller pins next to it. **These GND and VIN connections are required for each Motoron in a stack of Motorons.** Connecting two Motorons via stackable headers does not connect their VIN pins at all, and it does not the connect their GND pins in a way that is meant to carry the large currents involved in motor control. Note that connecting power to VIN does not power the Motoron’s microcontroller and does not cause any LEDs to turn on.

Each motor should have one lead connected to an MxA pin (M1A, M2A, or M3A) and the other lead connected to the MxB pin with the matching motor number. The Motoron’s concept of “forward” corresponds to MxA driving high while MxB drives low, so you might consider this when deciding which motor lead connects to which Motoron pin. You can also flip the wires later if you want to flip the direction of motion.

Connecting a controller


Motoron M3S256 shield being controlled by an Arduino Uno.


A Motoron M3H256 being controlled by a Raspberry Pi.

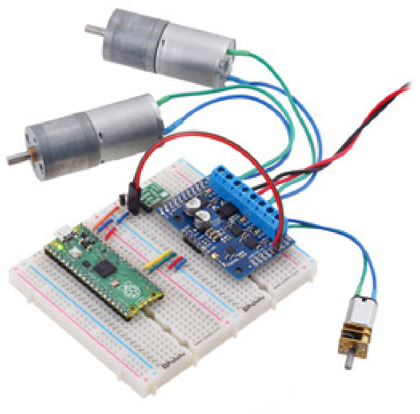
The Motoron M3S256 shield is designed to be plugged into the 1×10 and 1×8 female headers of an Arduino or Arduino-compatible board that has the shape of the **Arduino Uno R3 [https://www.pololu.com/product/2191]** using stackable female headers or male headers soldered to the Motoron. Similarly, the Motoron M3H256 is designed to be plugged into pins 1 through 20 of a Raspberry Pi using female headers.

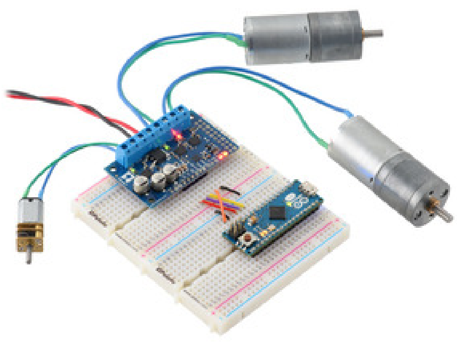
Plugging the Motoron into a controller this way connects the GND, SDA, and SCL pins of both boards, allowing the controller to communicate with the Motoron via I²C. It also powers the Motoron's microcontroller by connecting the Arduino's IOREF or the Raspberry Pi's 3V3 pin to the Motoron's logic voltage.

The Motoron does not need to be connected directly to the controller: it can be connected through another board (including other Motoron boards) as long as those boards pass the GND, SCL, SDA, and logic voltage connections through.

You can also connect the Motoron to a controller board that has a different shape as long as you make the same connections. The Motoron's GND, SCL, and SDA pins should be connected to the corresponding pins on the controller board, and the Motoron's logic voltage (labeled IOREF or 3V3 depending on which type of Motoron you have) should be connected to the logic voltage supply of the controller board, which should be between 2.8 V and 5.5 V.

Most of the pins on the Motoron have a spacing of 0.1" and are on the same 0.1" grid, making it easy to connect the Motoron to a perf-board or breadboard.


A Raspberry Pi Pico on a breadboard using a Motoron M3S256 shield to control three motors.


An Arduino Micro on a breadboard using a Motoron M3H256 to control three motors.

After you have connected one Motoron to a microcontroller, you can connect other Motorons to the same microcontroller simply by stacking them above or below the first one.

Once you make the GND and logic power connections and turn on the logic power, you should see the Motoron's yellow LED blink. The red LED will also turn on unless something is communicating with the Motoron and causing it to turn the LED off.

Powering the controller

By default, the Motoron does not supply power to the Arduino or Raspberry Pi, so you will need to power them separately. However, there are options for powering the controllers, as documented in **Section 4** for the M3S256, and in **Section 5** for the M3H256.

3.3. Enabling I²C on the Raspberry Pi

This section explains how to enable the correct I²C bus on your Raspberry Pi, make sure that your user has permission to access it, and test your setup.

The Motoron is designed to connect to the I2C1 bus on the Raspberry Pi, which uses GPIO pin 2 for SDA and GPIO pin 3 for SCL. If you are using Raspberry Pi OS, this bus is represented by `/dev/i2c-1` : that is the name of the device node that programs on your Raspberry Pi will open in order to communicate with the Motoron or any other targets on the bus.

Try typing `ls /dev/i2c*` to list your system's available I²C busses. If `/dev/i2c-1` is not in the list then you should run `sudo raspi-config nonint do_i2c 0` to enable it. You must reboot your Raspberry Pi for this change to take effect.

We recommend adding your user to the `i2c` group so you can access the Motoron and other I²C devices without using `sudo` . Run the `groups` command to see what groups your user belongs to.

If `i2c` is not in the list, then you should add your user to it by running `sudo usermod -a -G i2c $(whoami)`, logging out, and then logging in again.

After you have enabled I²C and connected your Motoron to your Raspberry Pi's I²C bus, run `i2cdetect -y 1`. If everything is set up correctly, you should see output like this:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

This output means that the Raspberry Pi detected a device at address 16 (0x10 in hex), which is the default I²C address used by the Motoron.

3.4. Setting I²C addresses with a Raspberry Pi

Each device on an I²C bus should have a unique address so that you can communicate with the device without interfering with other devices on the bus. By default, the Motoron uses I²C address 16, so if you have connected two or more Motorons to your bus, or you have another device that uses address 16, you will need to change the I²C addresses of one or more Motorons.

Warning: If you have devices on your I²C bus that are not Motorons, the procedure below could cause undesired behavior when it sends commands to them that are intended for the Motorons.

The recommended procedure for setting the I²C address of one or more Motorons that are connected to the I²C bus of your Raspberry Pi is:

1. Ensure that the JMP1 pin on each Motoron is not connected to anything.
2. Download the **Motoron Motor Controller Python library for Raspberry Pi [https://github.com/pololu/motoron-rpi]**, and install its dependencies, as described in the "Getting started" section of its README file.
3. In a Terminal, use `cd` to navigate into the directory holding the library and its examples.
4. Run `./set_i2c_addresses_example.py`. This utility will print some information and then prompt you for a command.
5. Perform a scan of the I²C bus by typing "s" followed by "Enter". You should get output that looks like this:

```
Scanning for I2C devices…
Found device at address 0
Found device at address 16
Done.
```

The scan detects that a device on the bus is responding to address 16 because that is the Motoron’s default address. It also detects a device on address 0 because that is the I²C general call address and all Motorons respond to it by default, in addition to the normal address. We will use address 0 to send commands in later steps, so if the scan does not detect any devices on address 0, those steps will probably fail.

6. Connect the JMP1 pin of one Motoron to GND. If you have not soldered headers to the JMP1 pin and its adjacent GND pin, you can connect those two pins together using a wire, test clip, or with a **shorting block [https://www.pololu.com/product/968]** attached to a 1×2 **male header [https://www.pololu.com/product/965]**. Connecting JMP1 to GND is how we select which Motoron’s address will be changed in the next step. Only one Motoron at a time should have its JMP1 pin connected to GND.
7. Set the address of the selected Motoron by typing “a”, followed by the address (in decimal), and then “Enter”. We recommend picking an address between 8 and 119 that is not in use by any other devices on your bus (numbers outside that range could work too, but they are reserved for other uses by the I²C specification). For example, type “a17” to set the address of the Motoron to 17. Alternatively, you can just send “a” by itself in order to have the program automatically pick an address for you, starting at 17 and skipping addresses that are already in use on the bus.

   This sends a “Write EEPROM” command to all of the Motorons using address 0, but the command will only have an effect on the one Motoron whose JMP1 line is low. That Motoron will record the address in its non-volatile EEPROM memory, but will not start using it yet.
8. Disconnect the JMP1 pin from GND.
9. Repeat the last three steps for every Motoron whose address you wish to change.
10. Type “r” to make the new addresses take effect. This sends a “Reset” command to all of the Motorons using address 0. Alternatively, you can power cycle the system or use the RST pin to reset the Motorons.
11. Perform another scan of the I²C bus by typing “s”. Check that a device is now found on each of the addresses that you assigned.

3.5. Setting I²C addresses with an Arduino

Each device on an I²C bus should have a unique address so that you can communicate with the device without interfering with other devices on the bus. By default, the Motoron uses I²C address 16, so if you have connected two or more Motorons to your bus, or you have another device that uses address

16, you will need to change the I²C addresses of one or more Motorons.

Warning: If you have devices on your I²C bus that are not Motorons, the procedure below could cause undesired behavior when it sends commands to them that are intended for the Motorons.

The recommended procedure for setting the I²C address of one or more Motorons that are connected to the I²C bus of your Arduino is:

1. Ensure that the JMP1 pin on each Motoron is not connected to anything.
2. Install the **Motoron Arduino library [https://github.com/pololu/motoron-arduino]** using the Arduino library manager. You can open the Library Manager from the “Tools” menu by selecting “Manage Libraries…”. If necessary, see the **library’s README [https://github.com/pololu/motoron-arduino]** for more information about how to install it.
3. Upload the **SetI2CAddresses** example to your Arduino. If the Motoron library is installed properly, you can find this example under Files > Examples > Motoron > SetI2CAddresses.
4. Open the Arduino IDE’s Serial Monitor, which you can find in the “Tools” menu.
5. Perform a scan of the I²C bus by typing “s” in the box at the top of the Serial Monitor and clicking “Send”. You should get output that looks something like this:

   ```
   Scanning for I2C devices…
   Found device at address 0
   Found device at address 16
   Done.
   ```

   The scan detects that a device on the bus is responding to address 16 because that is the Motoron’s default address. It also detects a device on address 0 because that is the I²C general call address and all Motorons respond to it by default, in addition to the normal address. We will use address 0 to send commands in later steps, so if the scan does not detect any devices on address 0, those steps will probably fail.
6. Connect the JMP1 pin of one Motoron to GND. If you have not soldered headers to the JMP1 pin and its adjacent GND pin, you can connect those two pins together using a wire, test clip, or with a **shorting block [https://www.pololu.com/product/968]** attached to a 1×2 **male header [https://www.pololu.com/product/965]**. Connecting JMP1 to GND is how we select which Motoron’s address will be changed in the next step. Only one Motoron at a time should have its JMP1 pin connected to GND.
7. Set the address of the selected Motoron by typing “a” in the box at the top of the Serial Monitor, followed by the address (in decimal), and clicking “Send”. We recommend picking an address between 8 and 119 that is not in use by any other devices on your bus

(numbers outside that range could work too, but they are reserved for other uses by the I²C specification). For example, type “a17” to set the address of the Motoron to 17. Alternatively, you can just send “a” by itself in order to have the sketch automatically pick an address for you, starting at 17 and skipping addresses that are already in use on the bus.

This sends a “Write EEPROM” command to all of the Motorons using address 0, but the command will only have an effect on the one Motoron whose JMP1 line is low. That Motoron will record the address in its non-volatile EEPROM memory, but will not start using it yet.

8. Disconnect the JMP1 pin from GND.
9. Repeat the last three steps for every Motoron whose address you wish to change.
10. Send “r” using the Serial Monitor to make the new addresses take effect. This sends a “Reset” command to all of the Motorons using address 0. Alternatively, you can power cycle the system or use the RST pin to reset the Motorons.
11. Perform another scan of the I²C bus by sending “s”. Check that a device is now found on each of the addresses that you assigned.

3.6. Writing code

This section documents what you need to know to get started writing code to control the Motoron.

The Motoron’s I²C interface allows you to send commands to it and receive responses from it. The commands are sequences of bytes (8-bit numbers from 0 to 255) and the responses are also sequences of bytes. The details of how the I²C interface works are documented in **Section 7**. The details of what commands are supported and how to encode them in bytes are documented in **Section 3.6**. Numbers prefixed with “0x” here are written in hexadecimal notation (base 16), and they are written with their most significant digits first, just like regular decimal numbers.

Arduino library and examples

If you are controlling the Motoron from an Arduino or Arduino-compatible board, we recommend that you install our **Motoron Arduino library** **[https://github.com/pololu/motoron-arduino]** and use one of the examples that comes with it as a starting point for your code. The library comes with these beginner-friendly examples, which you can find in the Arduino IDE under Files > Examples > Motoron.

- **Simple**: Shows how to control the Motoron in the simplest way.
- **Careful**: Shows how to shut down the motors whenever any problems are detected.
- **Robust**: Shows how to ignore or automatically recover from problems as much as possible.

Each of these examples just controls one Motoron. If you are using multiple Motorons, you can create an additional MotoronI2C object for each controller, and pass the I²C address of each Motoron to the constructor for each object. This can be done for any of the examples listed above, and the library comes with an example named **SimpleMulti** which is based on the Simple example and shows how to control multiple Motorons this way.

Python library and examples

If you are controlling the Motoron from a Raspberry Pi, you might consider downloading our **Motoron Python library [https://github.com/pololu/motoron-rpi]** and using one of the examples that comes with it as a starting point for your code. The Python library was designed to have the same features and behave nearly the same as the Arduino library. However, one major difference is that it generally throws an exception whenever there is a communication error, whereas the Arduino library only reports errors via its `getLastError()` method.

Like the Arduino library, the Python library comes with these beginner-friendly examples:

- **simple_example.py**: Shows how to control the Motoron in the simplest way.
- **careful_example.py**: Shows how to shut down the motors whenever any problems are detected.
- **robust_example.py**: Shows how to ignore or automatically recover from problems as much as possible.

Each of these examples just controls one Motoron. If you are using multiple Motorons, you can create an additional MotoronI2C object for each controller, and pass the I²C address of each Motoron to the constructor for each object using the `address` parameter. This can be done for any of the examples listed above, and the library comes with an example named **simple_multi_example.py** which is based on **simple_example.py** and shows how to control multiple Motorons this way.

The library also comes with an example called **simple_no_library_example.py** which is equivalent to **simple_example.py**, but does not use the Motoron library. This example is meant to be used as a reference for people trying to get started with the Motoron from a different programming environment, since it is easier to see exactly what bytes are being sent.

Initialization sequence

This is a sequence of commands you can run near the beginning of your code to help you get started with controlling the Motoron.

Description: Reinitialize
Bytes: 0x94 0x74

This command resets the Motoron to its default state (mostly). We recommend doing this when starting up to ensure that the behavior of your system will only depend on the code you are currently running, instead of being affected by settings that you might have set previously in an old version of your code. The bytes shown above are the command byte for the “Reinitialize” command followed by a cyclic redundancy check (CRC) byte.

Description: Disable CRC
Bytes: 0x8B 0x04 0x7B 0x43

This command disables the cyclic redundancy check (CRC) feature of the Motoron (documented in **Section 10**). The CRC feature is enabled by default to make the Motoron less likely to execute incorrect commands when there are communication problems. Disabling CRC makes it easier to get started writing code for the Motoron because you do not have to implement the CRC computation or append CRC bytes to each command you send. Once you have gotten your system to work, you might consider implementing CRC and removing this command to make things more robust. This is an example of the more general “Set protocol options” command, and the last byte shown above is a CRC byte.

Description: Clear reset flag
Bytes: 0xA9 0x00 0x04

This command clears (sets to 0) a bit in the Motoron called the “Reset flag”. This flag gets set to 1 after the Motoron powers on or experiences a reset, and with the default configuration it is considered to be an error, so it prevents the motors from running.This is an example of the more general “Clear latched status flags” command, and there is no CRC byte appended because we disabled the CRC feature above.

The reset flag exists to help prevent running motors with incorrect settings. In case the Motoron itself gets reset while your system is running, the Reset flag will be set and the motors will not run.

Initialization sequence (with CRC)

This is similar to the initialization sequence above, except it leaves CRC enabled.

Description: Reinitialize
Bytes: 0x94 0x74

Description: Clear reset flag
Bytes: 0xA9 0x00 0x04

The bytes at the end of each command are CRC bytes. Instead of hard coding those bytes, you should be able to calculate each one by applying the CRC algorithm to the command and data bytes immediately before it.

Command timeout

By default, the Motoron will turn off its motors if it has not received a valid command in the last 1.5 seconds. The details of the command timeout feature are documented in **Section 8**. This means that the motors will stop running if your microcontroller crashes, goes into programming/bootloader mode, or stops running your motor control code for any other reason. You can send a “Set variable” command to configure the timeout period or disable the feature altogether. Change the “Command timeout” variable if you want to change the amount of time it takes for the Motoron to time out, or change the “Error mask” variable if you want to disable the command timeout. The “Set variable” command and all other commands are documented in **Section 9**.

Motion parameters

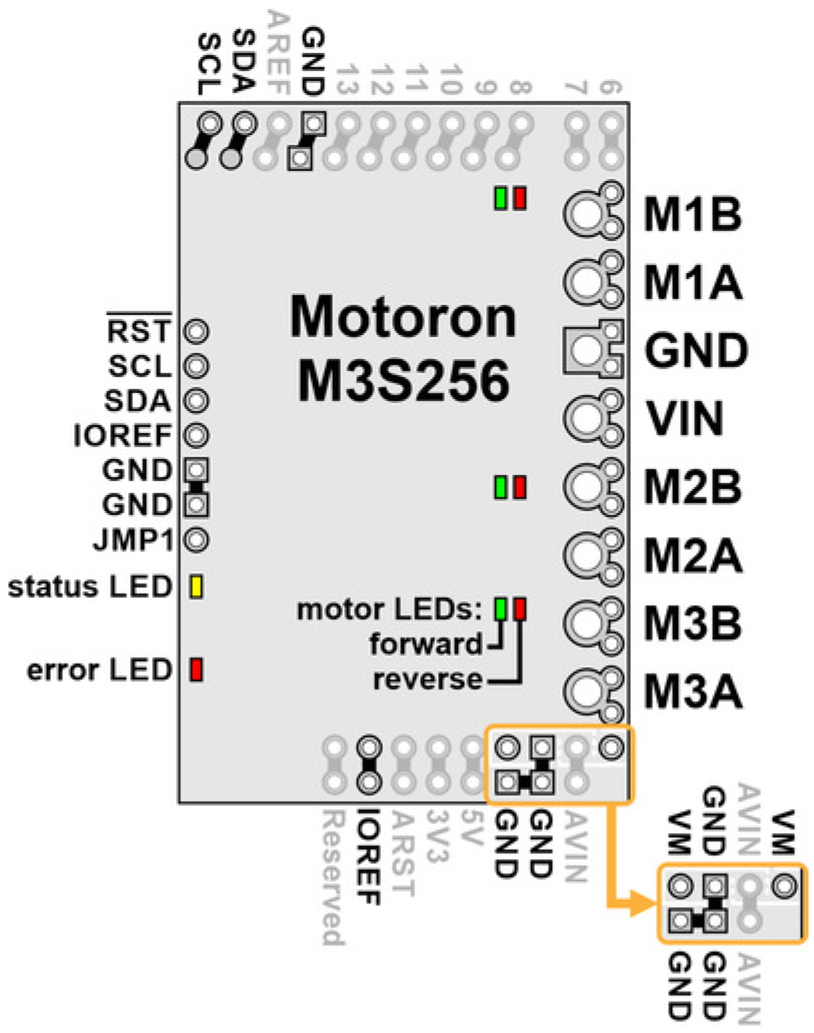
You can send a “Set variable” command (documented in **Section 9**) to configure how the motors move. In particular, you can set acceleration and deceleration limits for each motor and each direction of motion, which helps to reduce sudden current spikes or jerky motions.

The only Motoron setting that is stored in non-volatile memory is its I²C address. Every other setting, including the motion parameters and command timeout, gets reset to its default value whenever the Motoron powers on, resets, or receives a “Reinitialize” command.

Motor control

Once you have taken care of the initialization and configuration described above, you are ready to run some motors! See the “Set speed” and “Set all speeds” commands in **Section 9**.

4. Motoron M3S256 pinout



The diagram above identifies the control and power pins on the Motoron M3S256. Pins that are used by the motor controller are indicated in black, while pins that are not connected to anything by default are gray (these mostly serve as extra access points for the Arduino's pins if the Motoron is plugged in as a shield). **Section 3.2** explains how to connect motor power, motors, and a microcontroller to the Motoron.

The motor power supply should be connected to the **VIN** pin and adjacent **GND** pin, and a motor can be connected to each pair of **MxA** and **MxB** pins (e.g. M1A and M1B). For more information on choosing a power supply and motors, see **Section 3.1**.

The Motoron's logic is powered from the Arduino by the **IOREF** pin, and it is controlled via I²C through the **SCL** and **SDA** pins (see **Section 7**). Additional **GND** pins provide a common ground reference between the Motoron and Arduino.

The **JMP1** pin can be shorted to the adjacent GND pin to allow the Motoron’s I²C address to be changed, as detailed in **Section 3.5**. Also, shorting JMP1 to GND at startup causes the Motoron to ignore the address configured in EEPROM and use **15** as its I²C address instead.

The $\overline{\textbf{RST}}$ pin can be driven low to reset the Motoron; see **Section 11** for more details.

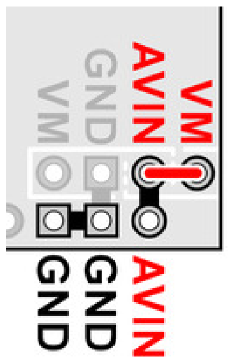
Powering the Arduino

The **VM** pins near the lower right corner of the board provide access to the reverse-protected motor supply voltage. VM can optionally be used to power the Arduino’s VIN pin (**AVIN**) either directly or through a regulator.

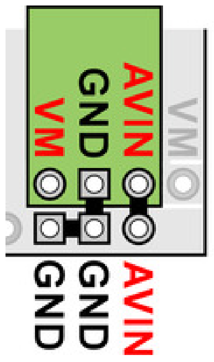
If the voltage of your motor power supply is within the allowed input voltage range for your Arduino, then you can power the Arduino by connecting the Motoron’s AVIN pin to the nearby VM pin. Doing this supplies power to Arduino’s VIN pin (AVIN) from the reverse-protected motor supply voltage (VM).

Alternatively, you can power the Arduino through a **voltage regulator** **[https://www.pololu.com/category/136/voltage-regulators]**. The Motoron M3S256 has VM, GND, and AVIN pins next to each other which are designed to be connected to a regulator. The regulator should be connected in the correct orientation so that the Motoron’s VM pin is connected to the regulator’s power input and the regulator’s power output is connected to AVIN. The motor power supply must be in the allowed input voltage range of the regulator, and the regulator must produce an output voltage that is within the allowed input voltage range of the Arduino. The regulator must also be able to supply enough current for the Arduino.

To avoid shorting two power outputs together, do not connect anything to the Arduino’s DC power jack while supplying power to AVIN through the Motoron.

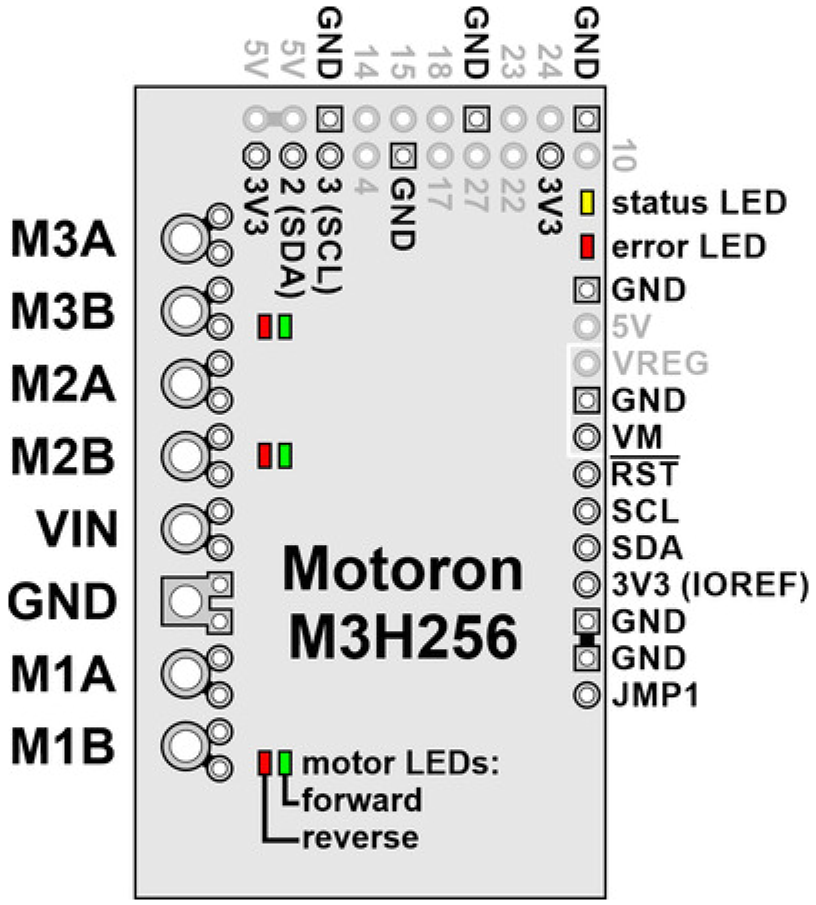


Powering the Arduino’s VIN from VM on a Motoron M3S256.



Powering the Arduino’s VIN from an external regulator connected to VM on a Motoron M3S256.

5. Motoron M3H256 pinout



The diagram above identifies the control and power pins on the Motoron M3H256. Pins that are used by the motor controller are indicated in black, while pins that are not connected to anything by default are gray.

The motor power supply should be connected to the **VIN** pin and adjacent **GND** pin, and a motor can be connected to each pair of **MxA** and **MxB** pins (e.g. M1A and M1B). For more information on choosing a power supply and motors, see **Section 3.1**.

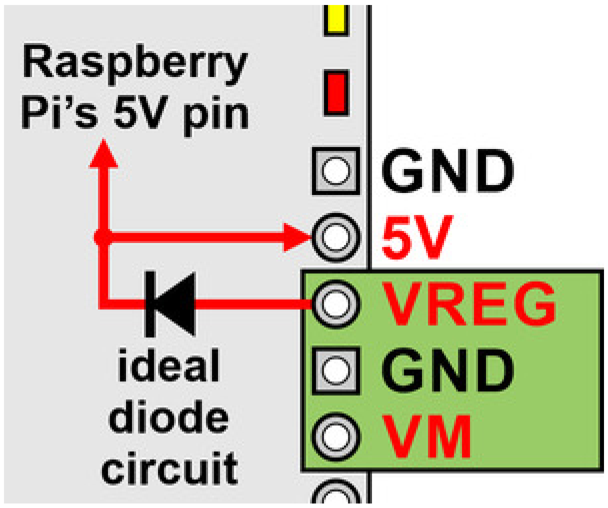
The Motoron's logic is powered from the **3V3** pin, which connects to the pin of the same name on the Raspberry Pi. The Motoron is controlled via I²C through the **SCL** and **SDA** pins (see **Section 7**). Additional **GND** pins provide a common ground reference between the Motoron and the Raspberry Pi.

The **JMP1** pin can be shorted to the adjacent GND pin to allow the Motoron's I²C address to be changed, as detailed in **Section 3.4**. Also, shorting JMP1 to GND at startup causes the Motoron to ignore the address configured in EEPROM and use **15** as its I²C address instead.

The $\overline{\textbf{RST}}$ pin can be driven low to reset the Motoron; see **Section 11** for more details.

The **VM** pin provides access to the reverse-protected motor supply voltage.

You can optionally power the Raspberry Pi by supplying 5 V to the **VREG** pin. To help achieve this, the Motoron M3H256 has adjacent VM, GND, and VREG pins which you can connect to a **voltage regulator** **[https://www.pololu.com/category/136/voltage-regulators]**. The regulator should be connected in the correct orientation so that the Motoron's VM pin connects to the regulator's power input and the regulator's power output connects to VREG. The motor power supply must be in the allowed input voltage range of the regulator, and the regulator must output 5 V (or something close enough to be tolerated by the Raspberry Pi). The regulator must also be able to supply enough current for the Raspberry Pi (e.g. 3 A for a Raspberry Pi 4). An ideal diode circuit on the Motoron prevents reverse current from flowing from the Raspberry Pi to the VREG pin if the Raspberry Pi is separately powered (for example, through its USB power receptacle). However, we do not recommend connecting external USB power to the Raspberry Pi while it is powered through the Motoron, since recent versions of the Raspberry Pi (including the 4 Model B and 3 Model B+) do not have a corresponding diode on their USB power input, so it is possible for the Motoron to backfeed a USB power adapter through the Raspberry Pi.



Powering the Raspberry Pi's 5V pin from an external regulator connected to VM on a Motoron M3H256.

The **5V** pin connects to the pin of the same name on the Raspberry Pi. It is also the output of the ideal diode circuit.

6. LED feedback

The Motoron Motor Controller has several LEDs to indicate its status.

Status LEDs

On the edge of the board, opposite the motor output pins, there are two status LEDs.

The **yellow status LED** indicates reset events and shows when the motor outputs are enabled.

- During the first half second after the Motoron has powered up or its processor has been reset, the yellow LED blinks 4 times.
- Otherwise, if the Motoron motor outputs are enabled or the Motoron is trying to enable them, the yellow LED is on solid. This corresponds to the "Motor output enabled" bit in the "Status flags" variable, which is documented in **Section 8**.
- Otherwise, if the Reset bit in the "Status flags" variable is set and it is configured to be an error, the yellow LED blinks for 0.5 s once per second. This is the default state, and it generally indicates that communication has not been established.
- Otherwise, the yellow LED blinks briefly once per second.

The **red error LED** indicates hardware issues or errors that prevent the motors from running.

- The red LED will be on solid if a motor fault is happening, motor power has been lost, or if there is a firmware-level error stopping the motors from running. More specifically, the red LED will be on if any of the "Motor faulting", "No power", or "Error active" flags documented in **Section 8** are 1.
- Otherwise the red LED will be off.

If the red LED is off, it does **not** necessarily mean that the VIN power voltage is high enough for the Motoron to drive motors.

Motor direction LEDs

On the side of the board with the motor output pins, each motor has two direction indicator LEDs.

The **green direction LED** indicates that the voltage on the MxA pin is high while the voltage on the MxB pin is low. This direction is called **forward** and corresponds to a positive speed numbers.

The **red direction LED** indicates that the voltage on the MxB pin is high while the voltage on the MxA pin is low. This direction is called **reverse** and corresponds to a negative speed numbers.

Both direction LEDs get brighter if the absolute value of the speed increases, or if the motor power supply (VIN) increases.

7. I²C interface

To control the Motoron, you will need to use its I²C interface.

I²C is a specification for a bus that can be used to connect multiple devices. The bus uses two signal lines: **SDA** is the data line and is used to transmit and receive data, while **SCL** is the clock line is used to coordinate the flow of data. There are two types of devices that can connect to an I²C bus: a *controller* is a device that initiates transfers of data, generates clock signals, and terminates transfers, while a *target* is a device that is addressed by a controller. The Motoron acts only as a target.

The Motoron’s SDA and SCL lines are each pulled up to the logic voltage of the Motoron (IOREF) with on-board 10 kΩ resistors.

I²C voltage levels

The voltages on SDA and SCL must not exceed the Motoron’s logic voltage by more than 0.3 V. Therefore, if the Motoron is running at 3.3 V, these pins are **not** 5 V tolerant. For the signals on these lines to be read properly by the Motoron, the low level must be less than 30% of the Motoron’s logic voltage, and the high level must be more than 70% of the Motoron’s logic voltage.

I²C clock speed

The Motoron’s I²C interface supports clock speeds up to 400 kHz. It uses clock stretching to slow down the transfer of data when bytes are written faster than it can handle, or if a read transfer is started before data is available.

If your I²C controller does not support clock stretching properly, you can avoid clock stretching by limiting your write transfers to be at most 31 bytes long and delaying for 1 millisecond after each write transfer to give the Motoron time to process it.

I²C address

By default, the Motoron uses the 7-bit I²C address **16**. This address is stored in the Motoron’s non-volatile EEPROM memory, and you can change it by sending a “Set EEPROM device number” command. If the JMP1 pin is shorted to GND at startup, the Motoron will ignore the address in EEPROM and use **15** as its I²C address instead. The Motoron determines what I²C address to use when it starts up, so any changes to the JMP1 pin or the EEPROM will not take effect until the next reset.

The Motoron also responds to the I²C general call address (0) in addition to its normal address by default. This allows you to send the same command to multiple Motoron targets simultaneously. The general call address is write-only; reading bytes from it is not supported. (However, if you send a command to the general call address that results in a response, you can read the response from an individual Motoron using its regular address.) You can use the “Set protocol options” command to disable the general call address, but it will become re-enabled the next time the Motoron is reset.

I²C protocol

There are two types of data transfers that can be initiated by an I²C controller: a *write* transfer writes some number of bytes to a target, and a *read* transfer reads some number of bytes from the target.

When you write bytes to the Motoron using write transfers, those bytes are interpreted as commands as described in **Section 9**. The Motoron does not care how the bytes are grouped into write transfers: you can send each command in its own transfer for simplicity, or send multiple commands together in a single transfer for extra efficiency. The Motoron acknowledges every byte written to it using I²C's built-in acknowledgment mechanism, regardless of whether those bytes actually form valid commands.

Some Motoron commands generate responses. To read the response to a command, you can start a read transfer after writing the last byte of a command, before you have written any other bytes. (To ensure that responses do not get mixed up, the Motoron clears its stored response every time a new byte is written.) It is OK to skip reading a response if you do not need it, or to just read part of it. It is also OK to read the response using multiple read transfers.

If you read bytes via I²C at a time when there is no response data available, the Motoron will provide a value of 0xAA for each byte you read. This can happen if you read at the wrong time, or if you read too many bytes. Enabling CRC for responses (as described in **Section 9**) and checking the value of the CRC byte is a good way to detect if this is happening.

8. Variable reference

The Motoron maintains a set of variables in RAM that contain information about its inputs, outputs, and status. The Motoron’s “Get variable” command allows you to read the variables, and there are other commands that allow you to set or modify the variables.

The variables are divided into two categories:

- **General** variables apply to all the motors, or the controller as a whole. The Motoron just stores one copy of each general variable.
- **Motor-specific** variables can be different for each motor. The Motoron stores a separate copy of the variable for each motor, and you must provide a motor number when accessing the variable.

This section lists all of the variables that the Motoron supports. In addition to the category, this section contains several pieces of information for each variable, if applicable:

- The **Offset** of each variable is its location among the variables in the same category. The offset is measured in bytes.
- The **Type** specifies how many bits the variable occupies, and says whether it is signed or unsigned.
- The **Range** specifies what values the variable can have.
- The **Default** specifies the value that the variable has when the controller starts up, before it has been modified.
- The **Units** specify the relationship between values of the variable and real-world quantities.
- The **Data** indicates how to interpret different possible values of the variable.
- The **Command** is the name of a command that can be used to set the variable.
- The **Arduino library** field shows the methods in the Arduino library that can be used to access the variable.

The term “bit 0” refers to the least significant bit of a variable (the bit that contributes a value of 1 to the variable when it is set). Accordingly, the other bits of a variable are numbered in order from least significant to most significant. All variables use little-endian byte ordering, meaning that the least-significant byte comes first.

List of variables

- **Protocol options**
- **Status flags**

- **VIN voltage**
- **Command timeout**
- **Error response**
- **Error mask**
- **Jumper state**
- **PWM mode**
- **Target speed**
- **Target brake amount**
- **Current speed**
- **Buffered speed**
- **Max acceleration forward**
- **Max acceleration reverse**
- **Max deceleration forward**
- **Max deceleration reverse**
- **Starting speed forward**
- **Starting speed reverse**
- **Direction change delay forward**
- **Direction change delay reverse**

Protocol options

Category	general
Offset	0
Type	unsigned 8-bit
Data	• **Bit 0:** CRC for commands • **Bit 1:** CRC for responses • **Bit 2:** I²C general call
Default	7 (all options enabled)
Command	Set protocol options

Arduino library	`void setProtocolOptions(uint8_t options)`
	`void enableCrc()`
	`void disableCrc()`
	`void enableCrcForCommands()`
	`void disableCrcForCommands()`
	`void enableCrcForResponses()`
	`void disableCrcForResponses()`
	`void enabeI2cGeneralCall()`
	`void disableI2cGeneralCall()`

This variable holds bits specifying which features of the Motoron’s command protocol are enabled. A bit value of 1 indicates that the corresponding feature is enabled. For more information about these features, see the documentation of the “Set protocol options” command in **Section 9**.

Status flags

Category	general
Offset	1
Type	unsigned 16-bit
Default	0x2200 (Reset, Error active)
Data	• **Bit 0:** Protocol error • **Bit 1:** CRC error • **Bit 2:** Command timeout latched • **Bit 3:** Motor fault latched • **Bit 4:** No power latched • **Bit 9:** Reset • **Bit 10:** Command timeout • **Bit 11:** Motor faulting • **Bit 12:** No power • **Bit 13:** Error active • **Bit 14:** Motor output enabled • **Bit 15:** Motor driving
Command	Clear latched status flags Set latched status flags

Arduino library	`uint16_t getStatusFlags()`
	`bool getProtocolErrorFlag()`
	`bool getCrcErrorFlag()`
	`bool getCommandTimeoutLatchedFlag()`
	`bool getMotorFaultLatchedFlag()`
	`bool getNoPowerLatchedFlag()`
	`bool getResetFlag()`
	`bool getMotorFaultingFlag()`
	`bool getNoPowerFlag()`
	`bool getErrorActiveFlag()`
	`bool getMotorOutputEnabledFlag()`
	`bool getMotorDrivingFlag()`
	`void clearLatchedStatusFlags(uint16_t flags)`
	`void clearResetFlag()`
	`void setLatchedStatusFlags(uint16_t flags)`

There are several status flags that are *latched*, meaning that after they get set to 1, they stay set until they are cleared by a "Clear latched status flags" command. Most of these flags are also cleared by the Reinitialize command.

- The **Protocol error** flag indicates that the Motoron received an invalid byte in a command other than the CRC byte, as documented in **Section 9**.
- The **CRC error** flag indicates that CRC for commands was enabled and the Motoron received an incorrect CRC byte at the end of a command.
- The **Command timeout latched** flag indicates that the Motoron's command timeout feature was activated because too much time has passed since it received a valid command. See the description of the **Command timeout** variable below for more information about this feature. This flag is the latched version of the "Command timeout" flag documented below.
- The **Motor fault latched** flag indicates that one or more of the motors experienced an error. This flag gets set to 1 whenever the "Motor faulting" flag below is 1, so see the documentation of that flag for more details.
- The **No power latched** flag indicates that the VIN voltage fell to a level that was definitely too low to run motors. This flag gets set to 1 whenever the "No power" flag documented below is 1, so see the documentation of that flag for more details.
- The **Reset** flag gets set to 1 when the Motoron powers on, or its processor is reset, or it receives a Reinitialize command.

There are several non-latched status flags which cannot be directly set or cleared.

- The **Command timeout** flag indicates that the Motoron's command timeout feature is active because too much time has passed since it received a valid command. Every valid command clears this bit. See the description of the **Command timeout** variable below for more information about this feature.
- The **Motor faulting** flag is 1 if one or more of the motors is experiencing an error. This means that something is going wrong with the motor, and the outputs for it will be disabled until the problem is resolved. Due to hardware limitations, there is no way to tell which motor is experiencing the fault.
  - For the Motoron M3S256 and M3H256, a fault occurs if the VIN power drops below about 4.4 V and remains above about 3.4 V. A fault occurs if a motor current over 8 A or a temperature over 165 °C is measured. The over-current and over-temperature faults are latched, so the motor will not recover from those errors until you use the "Clear motor fault" command, command the motors to coast, or disconnect motor power. There are other hardware

conditions that can cause temporary faults.

- The **No power** flag is 1 if the **VIN voltage** measurement indicates that the controller's VIN voltage is definitely too low to run the motors. If this flag is 0, the VIN voltage *might* be sufficient.
    - For the Motoron M3S256 and M3H256, this flag is 1 if and only if the VIN voltage measurement is less than 27. This corresponds to a VIN of 2.9 V if IOREF is 5 V, and 1.9 V if IOREF is 3.3 V.
- The **Error active** flag is 1 if there is a firmware-level error that is causing the Motoron to stop its motors. By default, the only two errors are the "Reset" flag and the "Command timeout" flags (documented above), but you can change which flags are considered to be errors by changing the **Error mask** variable documented below.
- The **Motor output enabled** flag is 1 if the Motoron is currently trying to enable any of its motor outputs. If this is 0, all of the motor outputs are disabled, meaning that the motors are coasting. If this is 1, some of the outputs might be enabled, but others might be disabled if there is inadequate VIN power or a motor fault is happening.
- The **Motor driving** flag is 1 if the Motoron is currently trying to drive any of its motors at a non-zero speed. This bit can only be 1 if the "Motor output enabled" flag is also 1.

VIN voltage

Category	general
Offset	3
Type	unsigned 16-bit
Range	0 to 1023
Units	IOREF × 0.02175

Arduino library	`void getVinVoltage()`
	`void getVinVoltageMv(uint16_t referenceMv)`

This two-byte variable holds a measurement of the voltage on the Motoron's VIN pin. It is a 10-bit ADC reading of the VIN voltage divided by 1047/47, with GND and IOREF used as the reference voltages.

If you are using our Arduino library, you can use the `getVinVoltageMv()` function to read this variable and convert it to millivolts.

The following C/C++ code shows how to take the raw value of this variable, along with the reference voltage (IOREF) in millivolts, and compute the VIN voltage in millivolts:

```
uint32_t vinVoltageMv = (uint32_t)vinVoltage * referenceMv / 1024 * 1047 / 47;
```

If you are using a 5 V Arduino, you can simply set `referenceMv` to 5000 in the code above. If you are using a 3.3 V Arduino, you can simply set `referenceMv` to 3300 in the code above.

Command timeout

Category	general
Offset	5
Type	unsigned 16-bit
Range	0 to 16250 (65 seconds)
Default	375 (1.5 seconds)
Units	4 ms
Command	Set variable

Arduino library	`void setCommandTimeoutMilliseconds(uint16_t ms)`
	`void getCommandTimeoutMilliseconds()`

The command timeout feature helps ensure that your motors will stop running if the device controlling the Motoron malfunctions, powers off, or gets disconnected.

The Motoron keeps track of how many milliseconds have passed since it received a valid command documented in **Section 9** that did not trigger a CRC error or a protocol error. If that time is greater than or equal to the time specified by the “Command timeout” variable, and the “Command timeout” variable is non-zero, then the Motoron will set two flags in the **Status flags** variable: “Command timeout” and “Command timeout latched”. When a valid command is received, the “Command timeout” flag gets cleared, and the “Command timeout latched” flag can be cleared with the “Clear latched status flags” command.

By default the non-latched command timeout flag is configured to be treated as an error, so when it is set, the each motor will decelerate to a stop and then coast.

This variable uses units of four milliseconds. For example, a value of 100 means the timeout will be 400 ms. However, if you set the variable using the `setCommandTimeoutMilliseconds` function in our Arduino library, you should specify the timeout in milliseconds and the Arduino library will take care of converting that value to correct units.

If you want to disable the command timeout feature, you can use the "Set variable" command to clear the command timeout bit in the "Error mask" variable.

Error response

Category	general
Offset	7
Type	unsigned 8-bit
Default	0 (Coast)
Data	• **0:** Coast • **1:** Brake • **2:** Coast now • **3:** Brake now
Command	Set variable

Arduino library	`void setErrorResponse(uint8_t response)`
	`void getErrorResponse()`

This variable defines how the Motoron will stop its motors when an error is happening. (The conditions that count as errors are defined by the **Error mask** variable.)

- **Coast** means that the Motoron will make all of its motors coast while obeying deceleration limits. This is equivalent to sending a "Set braking" command with a brake amount of 0 to each motor. If no deceleration limits are set, this is also equivalent to the "Coast now" error response.
- **Brake** means that the Motoron will make all of its motors brake while obeying deceleration limits. This is equivalent to sending a "Set braking" command with a brake amount of 800 to each motor. If no deceleration limits are set, this is also equivalent to the "Brake now" error response.
- **Coast now** means that the Motoron will make all of its motors coast immediately without obeying deceleration limits. This is equivalent to sending the "Coast now" command, and it is also equivalent to sending a "Set braking now" command with a brake amount of 0 to each motor.
- **Brake now** means that the Motoron will make all of its motors brake immediately without obeying deceleration limits. This is equivalent to sending a "Set braking now" command with a brake amount of 800 to each motor.

Error mask

Category	general
Offset	8
Range	0 to 0x7FF
Type	unsigned 16-bit
Default	0x600 (Command timeout and Reset)
Command	Set variable

Arduino library	`void setErrorMask()`
	`void disableCommandTimeout()`
	`void getErrorMask()`

This variable defines which status flags are considered to be errors. Each bit in this variable corresponds to the bit in the “Status flags” register in the same position. For example, bit 9 in this register corresponds to bit 9 in the “Status flags” register, which is the “Reset” bit.

The only flags that can be considered errors are the latching status flags (the flags that must be explicitly cleared before they will change to 0) and the “Command timeout” flag. If you try to set any other bits in the error mask to 1, those bits will be changed to 0.

Jumper state

Category	general
Offset	10
Type	unsigned 8-bit
Data	• **Bit 0:** JMP1 to GND jumper installed • **Bit 1:** JMP1 to GND jumper not installed Bits 2 to 7 are reserved and should each have a value of 1.
Arduino library	`void getJumperState()`

This variable indicates whether there is currently a jumper installed from the JMP1 pin to GND. This is determined with a digital reading on the JMP1 pin.

Bit 1 is always the logical inverse of bit 0, but if you read this variable and see that both bits are zero, it could mean that there are multiple Motorons using the same address, and they have different jumper states, and both of them are responding to your controller when you attempt to read this variable.

This variable can be useful as part of a procedure for verifying that every Motoron in your system has the correct address.

PWM mode

Category	motor-specific
Offset	1
Type	unsigned 8-bit
Data	Bits 0 to 3 specify the PWM frequency: • **0:** Default (20 kHz) • **1:** 1 kHz • **2:** 2 kHz • **3:** 4 kHz • **4:** 5 kHz • **5:** 10 kHz • **6:** 20 kHz • **7:** 40 kHz • **8:** 80 kHz Bits 4 to 7 are reserved and should be set to 0.
Default	0
Command	Set variable

Arduino library	`void setPwmMode(uint8_t motor, uint8_t mode)`
	`uint8_t getPwmMode(uint8_t motor)`

The lower 4 bits of this byte specify the PWM frequency to use for this motor.

You can set this variable with a “Set variable” command, which takes a 14-bit argument. The upper 6 bits are ignored, while the lower 8 bits get copied to this variable.

Due to hardware limitations on the M3S256 and M3H256, motors 2 and 3 must have the same PWM frequency, so setting the PWM mode of one of these motors also sets the PWM mode of the other.

Target speed

Category	motor-specific
Offset	2
Type	signed 16-bit
Range	−800 to 800
Default	0
Command	Set speed Set all speeds Set all speeds using buffers

Arduino library	`void setSpeed(uint8_t motor, int16_t speed)`
	`void setSpeedNow(uint8_t motor, int16_t speed)`
	`void setAllSpeeds(int16_t speed1, ...)`
	`void setAllSpeedsNow(int16_t speed1, ...)`
	`void setAllSpeedsUsingBuffers()`
	`void setAllSpeedsNowUsingBuffers()`
	`void getTargetSpeed(uint8_t motor)`

This is the speed at which the motor has been commanded to move. The “Current speed” (documented below) will move towards this over time, limited by the acceleration and deceleration limits (if enabled).

Target brake amount

Category	motor-specific
Offset	4
Type	unsigned 16-bit
Range	0 to 800
Default	0
Command	Set braking Set braking now Coast now

Arduino library	`void setBraking(uint8_t motor, uint16_t amount)`
	`void setBrakingNow(uint8_t motor, uint16_t amount)`
	`void coastNow()`
	`uint16_t getTargetBrakeAmount(uint8_t motor)`

This is the desired amount of braking to apply to the motors when the **current speed** is 0. A value of 0 corresponds to full coasting, while a value of 800 corresponds to full braking.

Due to hardware limitations, this amount of braking might not necessarily be applied when the current speed of the motor reaches zero.

For the Motoron M3S256 and M3H256: this type of Motoron is only capable of full coasting and full braking, and it can only use coasting if it coasts all the motors at once. Therefore, if the current speed of any of the motors is non-zero, or any of the motors has a non-zero target brake amount, the M3S256/M3H256 will not use coasting, and will instead use full braking for any motor whose current speed is zero. Also, the motor outputs might be inoperative due to inadequate VIN power or a motor fault, even if the Motoron is trying to apply full braking.

Current speed

Category	motor-specific
Offset	6
Type	signed 16-bit
Range	−800 to 800
Default	0
Command	Set speed Set all speeds Set all speeds using buffers

Arduino library	`void setSpeed(uint8_t motor, int16_t speed)`
	`void setSpeedNow(uint8_t motor, int16_t speed)`
	`void setAllSpeeds(int16_t speed1, ...)`
	`void setAllSpeedsNow(int16_t speed1, ...)`
	`void setAllSpeedsUsingBuffers()`
	`void setAllSpeedsNowUsingBuffers()`
	`void getCurrentSpeed(uint8_t motor)`

This is the speed that the Motoron is currently trying to apply to the motor.

- A value of 0 means the motor is braking (both outputs driving low), coasting (both outputs disabled), or something in between, as determined by the **target brake amount** variable and the hardware limitations of the controller.
- A value of 800 corresponds to the MxA output driving high (VIN) and the MxB output driving low (GND). This direction is called **forward**, and causes the **green** motor indicator LED to turn on.
- A value of −800 corresponds to the MxA output driving low (0 V) and the MxB output driving high (VIN). This direction is called **reverse**, and causes the **red** motor indicator LED to turn on.
- Intermediate values correspond to rapidly switching the motor outputs between braking and driving the motor in the specified direction.

Note: The “Current speed” variable only says the speed value that the Motoron is trying to apply to the motor. It is not based on any kind of sensor measurement. Also, the motor outputs might be inoperative due to inadequate VIN power or a motor fault even if the “Current speed” is non-zero,

Buffered speed

Category	motor-specific
Offset	8
Type	signed 16-bit
Range	−800 to 800, or −8192 for coasting
Default	0
Command	Set speed Set all speeds

Arduino library	`void setBufferedSpeed(uint8_t motor, int16_t speed)`
	`void setAllBufferedSpeeds(int16_t speed1, ...)`
	`void setAllSpeedsUsingBuffers()`
	`void setAllSpeedsNowUsingBuffers()`
	`void getCurrentSpeed(uint8_t motor)`

This is a speed that can be set ahead of time with the “Set speed” or “Set all speeds” commands. When you are ready to use the buffered speeds, you can use the “Set all speeds using buffers” command to make it actually take effect.

If you are using multiple Motoron controllers and want to change all of the motor speeds (nearly) instantaneously, the buffered speed feature can help you achieve that.

Max acceleration forward

Category	motor-specific
Offset	10
Type	unsigned 16-bit
Range	0 to 6400
Units	Speed change per 80 ms
Default	0
Command	Set variable

Arduino library	`void setMaxAccelerationForward(uint8_t motor, uint16_t accel)`
	`void setMaxAcceleration(uint8_t motor, uint16_t accel)`
	`uint16_t getMaxAccelerationForward(uint8_t motor)`

This variable specifies how quickly the motor’s current speed is allowed to increase when it is greater than or equal to 0. A value of 0 (the default) disables this acceleration limit, so the current speed can increase by any amount in a fraction of a millisecond. A non-zero value means that once every 10 ms, the current speed can increase by the specified value divided by 8. The Motoron keeps track of any fractional parts of the speed internally. Another way to think about this variable is that it is how much the current speed can change in 80 ms.

For example, if you set the max acceleration forward to 124, then the current speed can only increase by 15.5 speed units every 10 ms, or 124 speed units every 80 ms. This means it would take 520 ms (0.52 s) to accelerate from 0 (stopped) to 800 (full speed forward).

Note: Even if you set a maximum acceleration limit, the motor could experience abrupt acceleration if the current speed of the motor is non-zero while motor power is getting connected to the Motoron. To avoid this, you might want to set the “No power latched” bit in the **Error mask**.

Max acceleration reverse

Category	motor-specific
Offset	12
Type	unsigned 16-bit
Range	0 to 6400
Default	0
Command	Set variable
Arduino library	`void setMaxAccelerationReverse(uint8_t motor, uint16_t accel)` `void setMaxAcceleration(uint8_t motor, uint16_t accel)` `uint16_t getMaxAccelerationReverse(uint8_t motor)`

This is like **Max acceleration forward**, but for the reverse direction.

Max deceleration forward

Category	motor-specific
Offset	14
Type	unsigned 16-bit
Range	0 to 6400
Default	0
Command	Set variable
Arduino library	`void setMaxDecelerationForward(uint8_t motor, uint16_t decel)` `void setMaxDeceleration(uint8_t motor, uint16_t decel)` `uint16_t getMaxDecelerationForward(uint8_t motor)`

This variable specifies how quickly the motor’s current speed is allowed to decrease when it is greater than 0. A value of 0 (the default) disables this deceleration limit, so the current speed can decrease all the way to 0 in a fraction of millisecond. A non-zero value means that once every 10 ms, the current speed can decrease by the specified value divided by 8.

Max deceleration reverse

Category	motor-specific
Offset	16
Type	unsigned 16-bit
Range	0 to 6400
Default	0
Command	Set variable
Arduino library	`void setMaxDecelerationReverse(uint8_t motor, uint16_t decel)` `void setMaxDeceleration(uint8_t motor, uint16_t decel)` `uint16_t getMaxDecelerationReverse(uint8_t motor)`

This is like **Max deceleration forward**, but for the reverse direction.

Starting speed forward

Category	motor-specific
Offset	18
Type	unsigned 16-bit
Range	0 to 800
Default	0
Command	Set variable
Arduino library	`void setStartingSpeedForward(uint8_t motor, uint16_t speed)` `void setStartingSpeed(uint8_t motor, uint16_t speed)` `uint16_t getStartingSpeedForward(uint8_t motor)`

The Motoron allows the speed of the motor to accelerate instantly from 0 to the value of this variable, ignoring the **max acceleration forward**. This can be useful if you want your motor to accelerate faster by not spending any time accelerating through speeds that are too low to actually move the motor. This variable does not affect deceleration.

Starting speed reverse

Category	motor-specific
Offset	20
Type	unsigned 16-bit
Range	0 to 800
Default	0
Command	Set variable

Arduino library	`void setStartingSpeedReverse(uint8_t motor, uint16_t speed)`
	`void setStartingSpeed(uint8_t motor, uint16_t speed)`
	`uint16_t getStartingSpeedReverse(uint8_t motor)`

This is like **Starting speed forward**, but for the reverse direction. The Motoron allows the speed of the motor to accelerate instantly from 0 to the negated value of this variable, ignoring the **max acceleration reverse**.

Direction change delay forward

Category	motor-specific
Offset	22
Type	unsigned 8-bit
Range	0 to 250 (2500 ms)
Default	0
Units	10 ms
Command	Set variable

Arduino library	`void setDirectionChangeDelayForward(uint8_t motor, uint8_t duration)`
	`void setDirectionChangeDelay(uint8_t motor, uint8_t duration)`
	`uint8_t getDirectionChangeDelayForward(uint8_t motor)`

This variable specifies how long the Motoron should wait with the motor at speed 0 while switching directions from forward to reverse. For example, if the motor is currently driving in reverse (current speed less than 0), and the target speed is in the forward direction (positive), then the Motoron will decelerate the speed to 0, wait for an amount of time equal to the direction change delay forward, and then it will start accelerating in the forward direction (current speed greater than 0).

Direction change delay reverse

Category	motor-specific
Offset	23
Type	unsigned 8-bit
Range	0 to 250 (2500 ms)
Default	0
Units	10 ms
Command	Set variable
Arduino library	`void setDirectionChangeDelayReverse(uint8_t motor, uint8_t duration)` `void setDirectionChangeDelay(uint8_t motor, uint8_t duration)` `uint8_t getDirectionChangeDelayReverse(uint8_t motor)`

This is like **Direction change delay forward** but for the reverse direction.

9. Command reference

This section describes each of the commands supported by the Motoron Motor Controllers and how they are encoded as bytes on the I²C interface.

Each command begins with a byte called the command byte that has its most-significant bit set to 1. The command byte marks the start of the command and also indicates which command to execute. Some commands require additional bytes after the command byte, which are called data bytes and have their most-significant bits equal to 0. Unless you have disabled CRC for commands, the final byte of each command must be a CRC byte, which is calculated from the bytes came before it, as described in **Section 10**.

In the tables below that are labeled “Command encoding” and “Response encoding”, each cell of a table represents a single byte. CRC bytes are not shown in these tables, but each command requires a CRC byte at the end by default, and each response contains a CRC byte at the end by default. Numbers prefixed with “0x” are written in hexadecimal notation (base 16) and numbers prefixed with “0b” are written in binary notation. Numbers with these prefixes are written with their most significant digits first, just like regular decimal numbers.

The term “bit 0” refers to the least significant bit of a variable (the bit that contributes a value of 1 to the variable when it is set). Accordingly, the other bits of a variable are numbered in order from least significant to most significant.

For a reference implementation of the Motoron’s command protocol, see the **Motoron Arduino library** **[https://github.com/pololu/motoron-arduino]** or the **Motoron Python library** **[https://github.com/pololu/motoron-rpi]**.

List of commands

- **Get firmware version**
- **Set protocol options**
- **Read EEPROM**
- **Write EEPROM**
- **Reinitialize**
- **Reset**
- **Get variables**
- **Set variable**
- **Coast now**
- **Clear motor fault**

- **Clear latched status flags**
- **Set latched status flags**
- **Set speed**
- **Set all speeds**
- **Set all speeds using buffers**
- **Set braking**
- **Reset command timeout**

Get firmware version

Arguments	None
Response	Product ID and firmware version
Arduino library	`void getFirmwareVersion(uint16_t * productId, uint16_t * firmwareVersion)`

Command encoding:

0x87

Response encoding:

product ID low byte	product ID high byte	minor FW version (BCD format)	major FW version (BCD format)

Description:

This command generates a 4-byte response with identifying information about the firmware running on the device.

The first two bytes of the response are the low and high bytes of the product ID. The product ID 0x00CC corresponds to both the M3S256 and M3H256 (these models use the same firmware), and is encoded as 0xCC 0x00.

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]**. For example, 0x00 0x01 corresponds to firmware version 1.00.

Set protocol options

Arguments	**CRC for commands:** true or false **CRC for responses:** true or false **I²C general call:** true or false
Response	None

Arduino library	`void setProtocolOptions(uint8_t options)`
	`void enableCrc()`
	`void disableCrc()`
	`void enableCrcForCommands()`
	`void disableCrcForCommands()`
	`void enableCrcForResponses()`
	`void disableCrcForResponses()`
	`void enabeI2cGeneralCall()`
	`void disableI2cGeneralCall()`

Command encoding:

0x8B	protocol options byte	inverted protocol options byte

Description:

This command lets you change the Motoron’s protocol options. Each bit of the protocol options byte specifies whether to enable a particular feature.

- **Bit 0:** This bit should be 1 to enable CRC for commands and 0 to disable it. This feature is enabled by default and documented in **Section 10**.
- **Bit 1:** This bit should be 1 to enable CRC for responses and 0 to disable it. This feature is enabled by default and documented in **Section 10**.
- **Bit 2:** This bit should be 1 to enable the I²C general call address and 0 to disable it. This feature is enabled by default and documented in **Section 7**.

The other bits are reserved and should be set to 0. The effect of this command only lasts until the next time the Motoron loses power or its processor is reset, or it receives a Reinitialize command.

The second data byte should be equal to the protocol options byte but with the lower 7 bits all inverted. If it is some other value, the command fails and the Motoron reports a protocol error.

It is OK to provide a CRC byte at the end of this command even if CRC for commands has been disabled. For example, if you are not sure whether CRC for commands is enabled and you want to set the protocol options to 0x04 (disabling CRC but leaving the general call address enabled), send these four bytes: **0x8B 0x04 0x7B 0x43**. The fourth byte is the CRC byte. If you are want to set the protocol options to 0x00 (disabling CRC and the general call address), send these four bytes: **0x8B 0x00 0x7F 0x42**.

If you are using this command to enable or disable the I²C general call address, the command does not have an instant effect, so you might need to delay for 1 ms after sending the command.

Read EEPROM

Arguments	**Offset:** the address of the first byte to read, from 0 to 127 **Length:** the number of bytes to read, from 1 to 32
Response	The requested bytes

Arduino library	`void readEeprom(uint8_t offset, uint8_t length, uint8_t * buffer)`
	`uint8_t readEepromDeviceNumber()`

Command encoding:

0x93	offset	length

Description:

This command reads the specified bytes from the Motoron’s EEPROM memory, which is a 128-byte non-volatile memory that is used to store settings that persist through power interruptions and resets. See the “Write EEPROM” command below for more information about the settings stored in EEPROM.

Write EEPROM

Arguments	**Offset:** a number between 0 and 127 **Value:** a byte value between 0 and 255
Response	None

Arduino library	`void writeEeprom(uint8_t offset, uint8_t value)`
	`void writeEepromDeviceNumber(uint8_t number)`

Command encoding:

0x95	offset	lower 7 bits of value (0 to 127)	most significant bit of value (0 or 1)	first data byte (the offset) with lower 7 bits inverted	second data byte with lower 7 bits inverted	third data byte with lower 7 bits inverted

Description:

This command writes a value to one byte in the Motoron’s EEPROM memory, which is a 128-byte non-volatile memory that is used to store settings that persist through power interruptions and resets.

This command **only** works while the JMP1 pin is shorted to GND. If the JMP1 pin is not shorted to GND when this command is received, the EEPROM will not be modified.

This command takes about 5 ms to finish writing to the EEPROM. The Motoron’s microcontroller is stopped during this time, so it will not be able to respond to other commands or update its outputs. After running this command, we recommend waiting for at least 6 ms before you try to communicate with the Motoron.

Warning: Be careful not to write to the EEPROM in a fast loop. The EEPROM memory of the Motoron’s microcontroller is only rated for 100,000 erase/write cycles.

The only setting currently stored in the EEPROM memory is the **EEPROM device number**, a number between 0 and 127 that is stored at offset 1. The Motoron uses this number as its I²C address if it detects that JMP1 is not shorted to GND when it starts up. The default EEPROM device number is 16.

Although this command can write to any byte in EEPROM, we recommend that you only write to the EEPROM device number (offset 1). The byte at offset 0 is used internally by the Motoron, and the bytes at other offsets are reserved for use by new features in future firmware versions.

The first three data bytes after the command byte encode the offset and value. The last three data bytes are copies of the first three, but with the lower 7 bits inverted. If the third data byte is something other than 0 or 1, or the last three data bytes are incorrect, the Motoron reports a protocol error. The extra bytes in the command reduce the risk of accidental writes to the EEPROM.

Reinitialize

Arguments	None
Response	None
Arduino library	`void reinitialize()`

Command encoding:

0x96

Description:

This command resets most of the Motoron’s variables to their default state.

- The protocol options are reset to their default values (meaning that CRC and the I²C general call address is enabled).
- The latched status flags are cleared and the Reset flag is set to 1.
- The command timeout is reset to 250 (1000 ms).
- The error response and error mask are reset to their default values.
- The motors will start decelerating down to a speed of zero—respecting the previously-set deceleration limits—and then coast. This process can be interrupted by subsequent motor control commands (Coast, Set speed, Set all speeds, Set all speeds using buffers).
- The target speed, target brake amount, buffered speed, acceleration limits, deceleration limits, starting speeds, and direction change delays for each motor are reset to 0.

It is OK to provide a CRC byte at the end of this command even if CRC for commands has been disabled. For example, if you want to send the Reinitialize command and you are not sure whether CRC for commands is enabled, you can send the following two bytes: **0x96 0x74**. The second byte is the CRC byte.

Reset

Arguments	None
Response	None
Arduino library	`void reset()`

Command encoding:

0x99

Description:

This command causes a full hardware reset, and is equivalent to briefly driving the Motoron’s RST pin low. **The Motoron’s RST pin is briefly driven low** by the Motoron itself as a result of this command.

After running this command, we recommend waiting for at least 5 ms before you try to communicate with the Motoron.

Get variables

Arguments	**Motor:** a motor number, or 0 for general variables **Offset:** the address of the first variable to fetch **Length:** the number of bytes to fetch, from 1 to 32
Response	The requested bytes
Arduino library	`void getVariables(uint8_t motor, uint8_t offset, uint8_t length, uint8_t * buffer)` Several functions with names starting with `get`

Command encoding:

0x9A	motor	offset	length

Description:

This command fetches a range of bytes from the Motoron’s variables, which are stored in the Motoron’s RAM and represent the current state of the Motoron.

To fetch variables specific to a particular motor, set the **motor** argument to the motor number (between 1 and the number of motors supported by the Motoron). To fetch general variables applicable to all motors, set the **motor** argument to 0. The Motoron reports a protocol error if this argument is invalid.

The **offset** argument specifies the location of the first byte you want to fetch. The **length** argument specifies how many bytes to read. Each variable, along with its offset and size, is documented in **Section 8**.

It is OK to read past the last variable. The Motoron will return zeros when you try to read from unimplemented areas of the variable space.

All multi-byte variables retrieved by this command are returned in little-endian format, meaning that the least-significant byte comes first.

Set variable

Arguments	**Motor:** a motor number, or 0 for general variables **Offset:** the address of the variable to set (only certain offsets allowed) **Value:** the new number to store in the variable (14-bit)
Response	None
Arduino library	`void setVariable(uint8_t motor, uint8_t offset, uint16_t value)` Several functions with names starting with `set`

Command encoding:

0x9C	motor	offset	lower 7 bits of the value	bits 7 through 13 of the value

Description:

This command sets the value of the variable specified variable.

The **motor** and **offset** arguments specify which variable to set. These arguments are equivalent to the motor and offset arguments of the “Get variable” command. However, this command can only set certain variables, and the offset argument must point to the **first byte** of the variable. The Motoron will report a protocol error if the motor or offset arguments are invalid.

The **value** argument specifies the 14-bit number to set the variable to. The Motoron looks at all 14 bits of the value argument, even if the variable you are setting is 8-bit. If the value specified by those 14 bits is outside of the allowed range of values for the variable, the Motoron will change it to the closest allowed value before setting the variable.

Each variable, along with its offset, allowed range of values, and whether it can be set with this command, is documented in **Section 8**.

Here is some example C/C++ code that will generate the correct bytes, given integers `motor`, `offset`, and `value` and an array called `command`:

```
command[0] = 0x9C;  // Set Variable
command[1] = motor & 0x7F;
command[2] = offset & 0x7F;
command[3] = value & 0x7F;
command[4] = (value >> 7) & 0x7F;
```

Coast now

Arguments	None
Response	None
Arduino library	`void coastNow()`

Command encoding:

0xA5

Description:

This command causes all of the motors to immediately start coasting. For each motor, it sets the target brake amount, target speed, and current speed to 0.

Clear motor fault

Arguments	**Unconditional:** true or false
Response	None

Arduino library	`void clearMotorFault(uint8_t flags = 0)`
	`void clearMotorFaultUnconditional()`

Command encoding:

0xA6	Bit 0: unconditional

Description:

If any of the Motoron's motors are currently experiencing a fault (error), *or* the **unconditional** argument is true, this command attempts to recover from the faults.

For the Motoron M3S256 and M3H256, this command does not disrupt the operation of any motors that are operating normally.

If you send this command while the “Current speed” variable of any motor is non-zero, it could cause the Motoron to recover from a fault and suddenly start driving the motor at that speed.

Clear latched status flags

Arguments	**Flags:** 10-bit value
Response	None
Arduino library	`void clearLatchedStatusFlags(uint16_t flags)` `void clearResetFlag()`

Command encoding:

0xA9	lower 7 bits of flags	upper 3 bits of flags

Description:

For each bit in the **flags** argument that is 1, this command clears the corresponding bit in the “Status flags” variable, setting it to 0. The “Status flags” variable and all of its bits are documented in **Section 8**.

The Reset flag is particularly important to clear: it gets set to 1 after the Motoron powers on or experiences a reset, and it is considered to be an error by default, so it prevents the motors from running. Therefore, it is necessary to use this command to clear the Reset flag before you can get the motors running (or alternatively you can change the error mask).

We recommend that **immediately after** you clear the Reset flag, you should configure the Motoron’s motor settings and error response settings. That way, if the Motoron experiences an unexpected reset while your system is running, it will stop running its motors and it will not start them again until all the important settings have been configured.

The Reset flag is bit 9 in the “Status flags” variable. Therefore, to clear it, you would set the flags argument to 0x200. This would result in a command with the following three bytes (not including the CRC byte): **0xA9 0x00 0x04**.

Set latched status flags

Arguments	**Flags:** 10-bit value
Response	None
Arduino library	`void setLatchedStatusFlags(uint16_t flags)`

Command encoding:

0xAC	lower 7 bits of flags	upper 3 bits of flags

Description:

For each bit in the **flags** argument that is 1, this command sets the corresponding bit in the "Status flags" variable to 1. The "Status flags" variable and all of its bits are documented in **Section 8**.

Set speed

Arguments	**Mode:** normal, now, or buffered **Motor:** a motor number **Speed**: from −800 to 800
Response	None

Arduino library	`void setSpeed(uint8_t motor, int16_t speed)`
	`void setSpeedNow(uint8_t motor, int16_t speed)`
	`void setBufferedSpeed(uint8_t motor, int16_t speed)`

Command encoding:

0xD1 for normal mode 0xD2 for now mode 0xD4 for buffered mode	motor	lower 7 bits of speed	upper 7 bits of speed

Description:

The **motor** argument should be between 1 and the number of motors that your Motoron supports. If it is invalid, the Motoron reports a protocol error.

The **speed** argument should be a speed between −800 and 800. If the specified speed is outside this range, the Motoron will change it to the closest valid speed between −800 and 800. See the documentation of the "Current speed" variable in **Section 8** for more details about what the different speed values mean. The speed argument is encoded as a 14-bit **two's complement [https://en.wikipedia.org/wiki/Two%27s_complement]** number.

The **mode** specifies when and how to apply the speed:

- **Normal mode:** The motor's "Target speed" is changed. The "Current speed" will start moving towards the target speed, obeying acceleration and deceleration limits. The

"Target brake amount" is also set to 800.

- **Now mode:** The motor's "Target speed" and "Current speed" are changed, so the motor outputs will change immediately. The "Target brake amount" is also set to 800.
- **Buffered mode:** The motor's "Buffered speed" is set. You can use the "Set all speeds using buffers" command to make it take effect.

Here is some example C/C++ code that will generate the correct bytes, given integers `motor` and `speed`:

```
command[0] = 0xD1;  // Set Speed, normal mode
command[1] = motor & 0x7F;
command[2] = speed & 0x7F;
command[3] = (speed >> 7) & 0x7F;
```

Set all speeds

Arguments	**Mode:** normal, now, or buffered **Speed for each motor**: from −800 to 800
Response	None

Arduino library	`void setAllSpeeds(int16_t speed1, ...)`
	`void setAllSpeedsNow(int16_t speed1, ...)`
	`void setAllBufferedSpeeds(int16_t speed1, ...)`

Command encoding:

0xE1 for normal mode 0xE2 for now mode 0xE4 for buffered mode	lower 7 bits of speed 1	upper 7 bits of speed 1	…

Description:

This command is equivalent to the "Set speed" command, but it sets the speed of all the motors at the same time. This command takes one speed argument for each motor supported by the controller. Each speed argument is encoded as two bytes, using the same speed encoding as the "Set speed" command. The speeds are sent in order by motor number, starting with the speed for motor 1.

If CRC for commands is **not** enabled, it is OK to send extra speeds. They will be ignored since their most significant bit is 0.

If you send fewer speeds than the number of motors supported, the command will not be executed, and the Motoron will report a protocol error when you send the next command.

Set all speeds using buffers

Arguments	**Mode:** normal or now
Response	None
Arduino library	`void setAllSpeedsUsingBuffers()` `void setAllSpeedsNowUsingBuffers()`

Command encoding:

0xF0 for normal mode 0xF3 for now mode

Description:

This command applies the buffered speeds that were previously set with “Set speed” or “Set all speeds” commands in buffered mode.

The **mode** specifies how to apply the speed:

- **Normal mode:** Each motor’s “Target speed” is set equal to its buffered speed. Each motor’s “Current speed” will start moving towards this value, obeying acceleration and deceleration limits.
- **Now mode:** Each motor’s “Target speed” and “Current speed” are set equal to its buffered speed, so the motor outputs will change immediately.

This command also sets each motor’s “Target brake amount” to 800.

This command does not change the buffered speeds.

Set braking

Arguments	**Mode:** normal or now **Motor:** a motor number **Brake amount**: from 0 to 800
Response	None

Arduino library	`void setBraking(uint8_t motor, uint16_t amount)`
	`void setBrakingNow(uint8_t motor, uint16_t amount)`

Command encoding:

0xB1 for normal mode 0xB2 for now mode	motor	lower 7 bits of brake amount	upper 7 bits of brake amount

Description:

The **motor** argument should be between 1 and the number of motors that your Motoron supports. If it is invalid, the Motoron reports a protocol error.

The **brake amount** argument should be a number between 0 and 800. If it is larger than 800, the Motoron will change it to 800. This command sets the "Target brake amount" variable of the specified motor to the value of this argument. A value of 0 corresponds to full coasting, while a value of 800 corresponds to full braking. However, due to hardware limitations, the resulting brake amount might be different from what is specified. See the documentation of the "Target brake amount" variable in **Section 8** for more details.

The **mode** argument specifies when and how to apply the specified brake amount:

- **Normal mode:** The motor's "Target speed" is set to 0. The "Current speed" will start moving towards the target speed, obeying deceleration limits. When it reaches zero, the specified brake amount will be used.
- **Now mode:** The motor's "Target speed" and "Current speed" are set to 0 so the desired brake amount will be applied immediately.

Here is some example C/C++ code that will generate the correct bytes, given integers `motor` and `amount`:

```
command[0] = 0xB1;  // Set braking, normal mode
command[1] = motor & 0x7F;
command[2] = amount & 0x7F;
command[3] = (amount >> 7) & 0x7F;
```

Reset command timeout

Arguments	None
Response	None
Arduino library	`void resetCommandTimeout()`

Command encoding:

0xF5

Description:

This command resets the command timeout, which means that if the command timeout feature is enabled, this command prevents the timeout from occurring for some time. (The command timeout is also reset by every other command documented here.) The command timeout feature is documented in **Section 8**.

Treatment of unrecognized and invalid bytes

If the Motoron receives a byte with a most significant bit of 1 while it was expecting a data byte for a command, the command is canceled and the Motoron reports a serial protocol error.

If the Motoron receives a byte with a most significant bit of 1 that is not a recognized command byte, it will usually report a protocol error. However, bytes 0x80, 0xFE, and 0xFF are ignored, and 0xAA is a reserved command byte that should not be used during normal operation.

If the Motoron receives a byte with a most significant bit of 0 while it is not expecting a data byte for a command, it ignores the byte.

10. Cyclic redundancy check (CRC)

To help prevent communication errors, the Motoron by default requires a cyclic redundancy check (CRC) byte to be appended to each command it receives, and it also appends a CRC byte to each response it sends. If the CRC byte for a command is incorrect, the Motoron will ignore the command and set the “CRC error” status flag. You can disable CRC by sending a “Set protocol options” command as documented in **Section 9**.

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 Motoron 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 that is tacked onto the end of a message.

The C code below shows one way to implement the CRC algorithm:

```c
#include <stdint.h>

uint8_t getCRC(uint8_t * message, uint8_t length)
{
  uint8_t crc = 0;
  for (uint8_t i = 0; i < length; i++)
  {
    crc ^= message[i];
    for (uint8_t j = 0; j < 8; j++)
    {
      if (crc & 1) { crc ^= 0x91; }
      crc >>= 1;
    }
  }
  return crc;
}
```

Note that the innermost for loop in the example above can be replaced with a lookup from a precomputed 256-byte lookup table, which should be faster.

For example, a “Set speed” command setting the speed of 1 to 100 with a CRC byte appended to it would be:

0xD1	0x01	0x64	0x00	0x68

11. Reset pin

The Motoron’s reset pin is labeled $\overline{\mathbf{RST}}$ on the board. This pin is normally pulled high, and driving this pin low resets the Motoron’s microcontroller. This pin is normally an input, but the Motoron does briefly drive it low when it receives a “Reset” command or if there is any other internal mechanism causing the Motoron to reset.