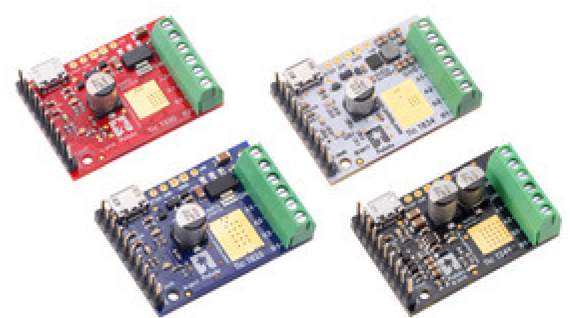
© 2001–2019 Pololu Corporation

Tic Stepper Motor Controller User’s Guide



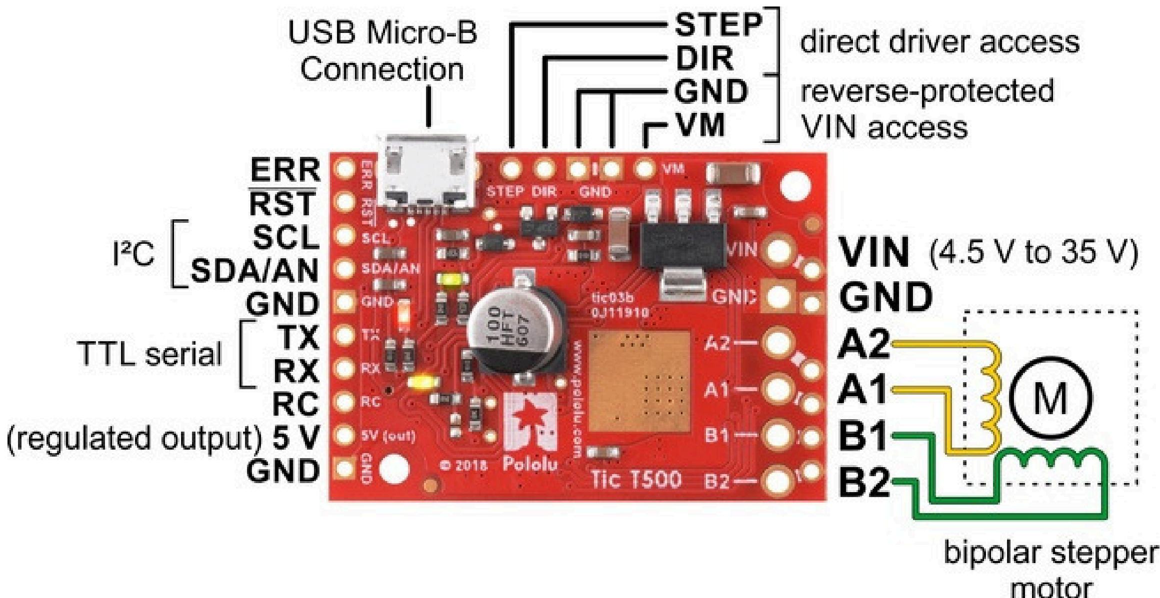
1. Overview 4
 1.1. Available versions 11
 1.2. Supported operating systems 14
2. Contacting Pololu 15
3. Getting started 16
 3.1. Installing Windows drivers and software 16
 3.2. Installing Linux software 18
 3.3. Installing macOS software 21
 3.4. LED feedback 22
4. Setting up the controller 26
 4.1. Choosing the power supply, Tic, and stepper motor 26
 4.2. Connecting the stepper motor and power supply 29
 4.3. Configuring and testing the stepper motor 31
 4.4. Setting up USB control 39
 4.5. Setting up serial control 41
 4.6. Setting up I²C control 45
 4.7. Setting up RC position control 49
 4.8. Setting up RC speed control 51
 4.9. Setting up analog position control 54
 4.10. Setting up analog speed control 56
 4.11. Setting up encoder position control 58
 4.12. Setting up encoder speed control 60
 4.13. Setting up STEP/DIR control 63
 4.14. Setting up limit switches and homing 64
5. Details 67
 5.1. Motion parameters 67
 5.2. Analog/RC input handling 70
 5.3. Encoder input handling 77
 5.4. Error handling 79
 5.5. Pin configuration 87
 5.6. Homing 91
 5.7. Upgrading firmware 92
 5.8. Logic power output (5V) 94
 5.9. Hardware design files 95
6. Setting reference 97
7. Variable reference 136
8. Command reference 146
9. Serial command encoding 162
10. I²C command encoding 169
11. USB command encoding 174

12. Writing PC software to control the Tic . . . 177
12.1. Example code to run ticcmd in C . . . 178
12.2. Example code to run ticcmd in Ruby . . . 179
12.3. Example code to run ticcmd in Python . . . 181
12.4. Running ticcmd with Windows shortcuts . . . 183
12.5. Example serial code for Linux and macOS in C . . . 186
12.6. Example serial code for Windows in C . . . 190
12.7. Example serial code in Python . . . 194
12.8. Example I²C code for Linux in C . . . 196
12.9. Example I²C code for Linux in Python . . . 199
12.10. Example code using the C API . . . 201
12.11. Example code using the C++ API . . . 203

© 2001–2019 Pololu Corporation

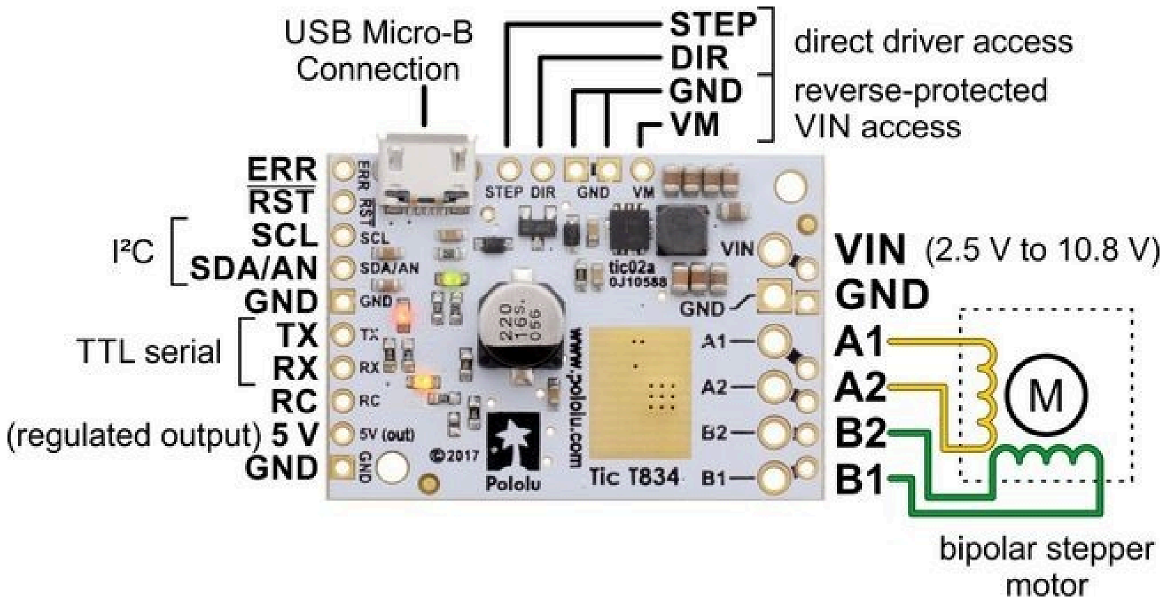
1. Overview

The Tic stepper motor controllers are a family of versatile, general-purpose modules designed to control one **bipolar stepper motor** **[https://www.pololu.com/category/87/stepper-motors]**. With a variety of supported interfaces–USB for direct connection to a computer, TTL serial and I²C for use with a microcontroller, RC hobby servo pulses for use in an RC system, analog voltages for use with a potentiometer or analog joystick, and quadrature encoder for use with a rotary encoder dial–and a wide array of configurable settings, the Tic controllers make it easy to add basic control of a bipolar stepper motor to a variety of projects. A free configuration utility (for Windows, Linux, and macOS) simplifies initial setup of the device and allows for in-system testing and monitoring of the controller via USB.

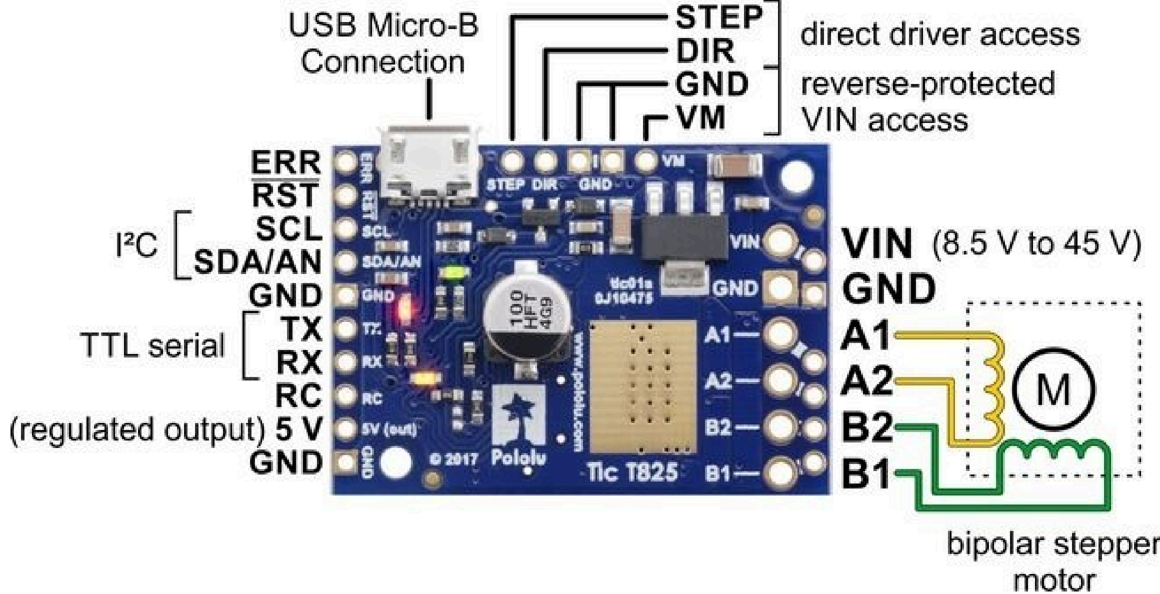


The **Tic T500**, shown above, is based on the MP6500 IC from Monolithic Power Systems (MPS). This driver IC supports microstepping with up to 8 microsteps per full step and features automatic decay mode selection, using internal current sensing to automatically adjust the decay mode as necessary to provide the smoothest current waveform. The Tic T500 can operate from 4.5 V to 35 V and features reverse-protection over the full input voltage range. It can deliver up to approximately 1.5 A continuous per phase without a heat sink or forced air flow (the peak current per phase is 2.5 A). The Tic T500's circuit board is red with white labels.

Powering the Tic T500 with a supply voltage between 4.5 V and 5.5 V might cause its logic voltage to be lower than normal, which could affect its operation. See **Section 4.1** for more information.

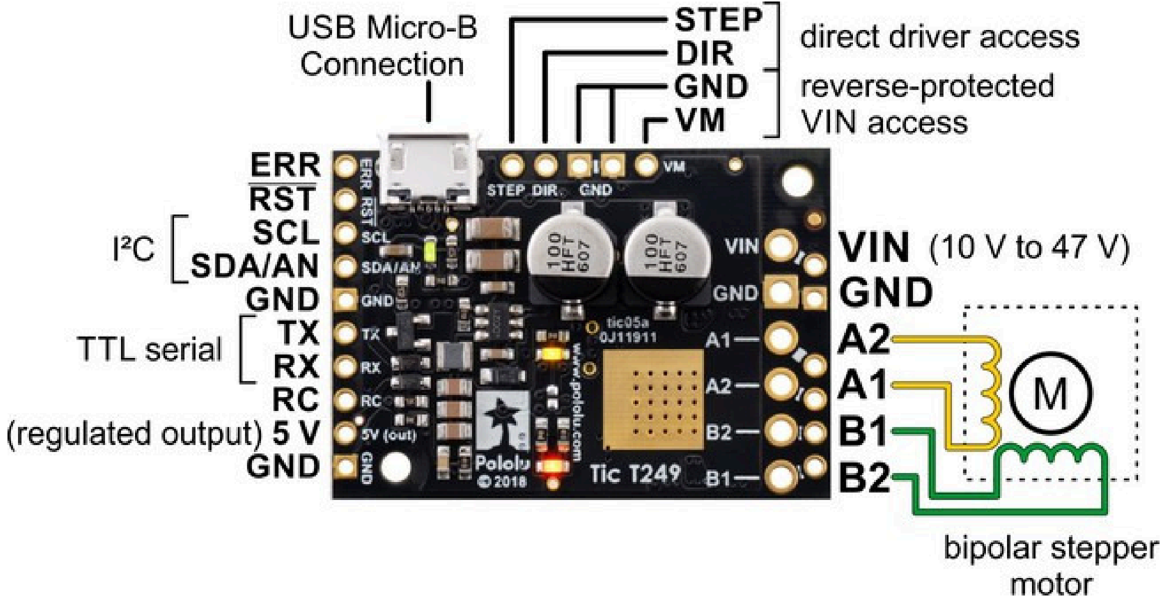


The **Tic T834**, shown above, is based on the DRV8834 IC from Texas Instruments. This driver IC supports microstepping with up to 32 microsteps per full step and features five configurable decay modes. The Tic T834 can operate from 2.5 V to 10.8 V and features reverse-voltage protection over the full input voltage range. It can deliver up to approximately 1.5 A per phase without a heat sink or forced air flow (absolute maximum is 2 A per phase). The Tic T834's circuit board is white with black labels.



The **Tic T825**, shown above, is based on the DRV8825 IC from Texas Instruments. This driver IC supports microstepping with up to 32 microsteps per full step and features three configurable decay

modes. The Tic T825 can operate from 8.5 V to 45 V and features reverse-voltage protection up to 40 V. It can deliver up to approximately 1.5 A per phase without a heat sink or forced air flow (absolute maximum is 2.5 A per phase). The Tic T825's circuit board is blue with white labels.



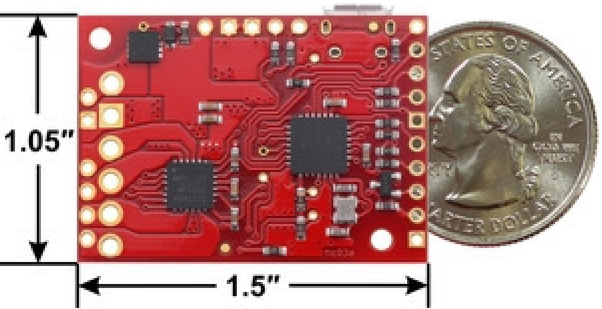
The **Tic T249**, shown above, is based on the TB67S249FTG IC from Toshiba. This driver IC supports microstepping with up to 32 microsteps per full step and offers several unique and innovative features. One of these is Toshiba's Active Gain Control (AGC), which automatically reduces the stepper motor current below the set limit based on the actual load on the motor, allowing for reduced unnecessary heat generation and higher peak power when the motor actually needs it. Another is Toshiba's Advanced Dynamic Mixed Decay (ADMD) technology, which dynamically switches between slow and fast decay modes based on the actual motor current, providing higher efficiency and smoother steps at high speed than you get with the traditional timing-based mixed decay. The Tic T249 can operate from 10 V to 47 V and features reverse-voltage protection up to 40 V. It can deliver up to approximately 1.8 A per phase without a heat sink or forced air flow (absolute maximum is 4.5 A per phase). The Tic T249's circuit board is black with white labels.

The table below lists the members of the Tic family and shows the key differences between them.

	Tic T500	Tic T834	Tic T825	Tic T249
Operating voltage range:	4.5 V to 35 V[(1)]	2.5 V to 10.8 V	8.5 V to 45 V[(1)]	10 V to 47 V[(1)]
Max continuous current per phase (no additional cooling):	1.5 A	1.5 A	1.5 A	1.8 A
Peak current per phase (additional cooling required):	2.5 A	2 A	2.5 A	4.5 A
Microstep resolutions:	full half 1/4 1/8	full half 1/4 1/8 1/16 1/32	full half 1/4 1/8 1/16 1/32	full half 1/4 1/8 1/16 1/32
Automatic decay selection:	✓			✓
Automatic gain control (AGC):				✓
Driver IC:	MP6500	DRV8834	DRV8825	TB67S249FTG
Price (connectors not soldered):	$19.95	$29.95	$29.95	$39.95
Price (connectors soldered):	$21.95	$31.95	$31.95	$41.95

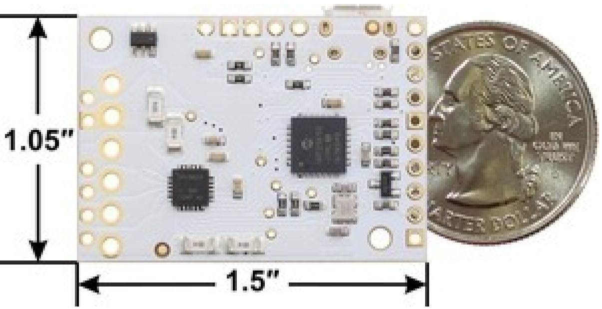
1 See product pages and user's guide for operating voltage limitations.

Features and specifications

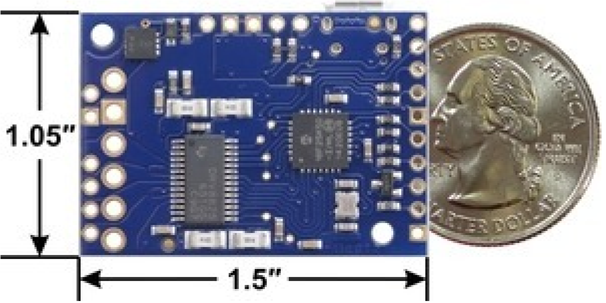


Tic T500 USB Multi-Interface Stepper Motor Controller, bottom view with dimensions. This picture shows the original tic03a version that shipped prior to 3 January 2019.

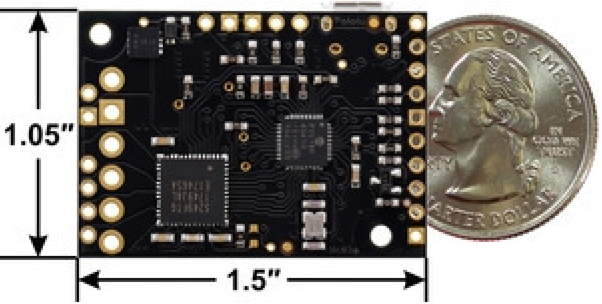
- Open-loop speed or position control of one bipolar stepper motor
- A variety of control interfaces:
    - **USB** for direct connection to a computer
    - **TTL serial** operating at 5 V for use with a microcontroller
    - **I²C** for use with a microcontroller
    - **RC hobby servo pulses** for use in an RC system
    - **Analog voltage** for use with a potentiometer or analog joystick
    - **Quadrature encoder** input for use with a rotary encoder dial, allowing full rotation without limits (*not* for position feedback)
    - **STEP/DIR** inputs for compatibility with existing stepper motor control firmware
- Acceleration and deceleration limiting
- Maximum stepper speed: 50,000 steps per second
- Very slow speeds down to 1 step every 200 seconds (or 1 step every 1428 seconds with reduced resolution).
- Up to six different microstep resolutions:
    - The Tic T825, Tic T834, and Tic T249 support full step, half step, 1/4 step, 1/8 step, 1/16 step, and 1/32 step



Tic T834 USB Multi-Interface Stepper Motor Controller, bottom view with dimensions.



Tic T825 USB Multi-Interface Stepper Motor Controller, bottom view with dimensions.

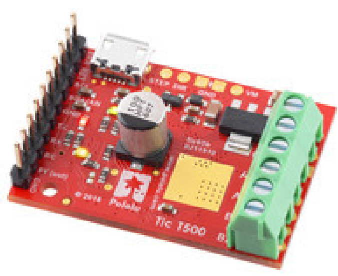


Tic T249 USB Multi-Interface Stepper Motor Controller, bottom view with dimensions.

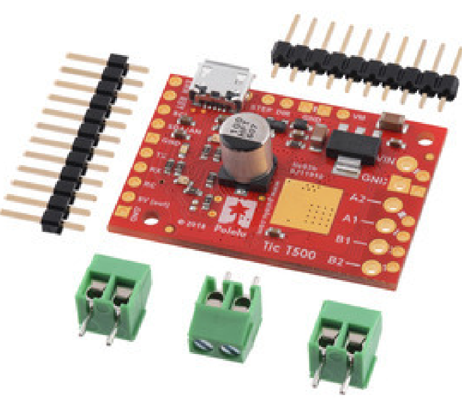
    - The Tic T500 supports full step, half step, 1/4 step, 1/8 step
- Digitally adjustable current limit
- Decay modes:
    - The Tic T500 features automatic decay mode selection.
    - The Tic T834 features 5 digitally configurable decay modes.
    - The Tic T825 features 3 digitally configurable decay modes.
    - The Tic T249 features Toshiba’s Advanced Dynamic Mixed Decay (ADMD) technology.
- Optional safety controls to avoid unexpectedly powering the motor
- Input calibration (learning) and adjustable scaling degree for analog and RC signals
- 5 V regulator (no external logic voltage supply needed)
- Optional limit switch inputs with homing capabilities
- Optional kill switch inputs
- STEP/DIR outputs for controlling external stepper motor drivers
- Connects to a computer through USB via a **USB A to Micro-B cable** **[https://www.pololu.com/product/2072]** (not included)
- Free configuration software available for Windows, Linux, and macOS

© 2001–2019 Pololu Corporation

1.1. Available versions

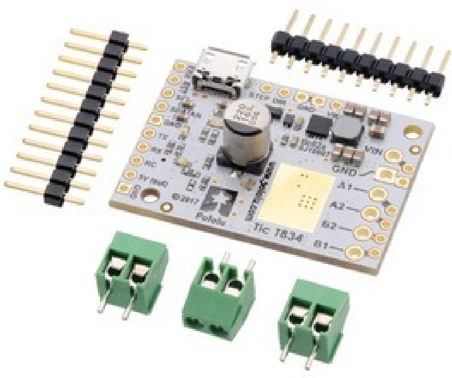


Tic T500 USB Multi-Interface Stepper Motor Controller (Connectors Soldered).



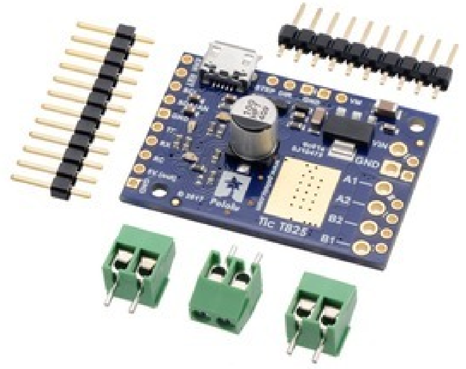
Tic T500 USB Multi-Interface Stepper Motor Controller (without connectors soldered) with included headers and terminal blocks.

Tic T834 USB Multi-Interface Stepper Motor Controller (Connectors Soldered).

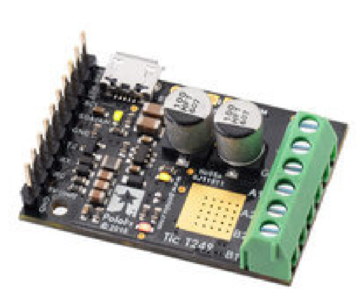


Tic T834 USB Multi-Interface Stepper Motor Controller (without connectors soldered) with included headers and terminal blocks.

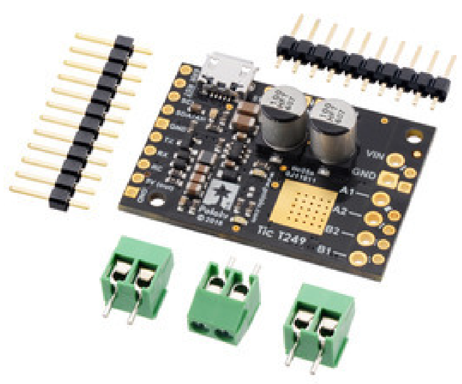
Tic T825 USB Multi-Interface Stepper Motor Controller (Connectors Soldered).



Tic T825 USB Multi-Interface Stepper Motor Controller (without connectors soldered) with included headers and terminal blocks.



Tic T249 USB Multi-Interface Stepper Motor Controller (Connectors Soldered).



Tic T249 USB Multi-Interface Stepper Motor Controller (without connectors soldered) with included headers and terminal blocks.

The Tic family consists of three different controllers: the **Tic T500**, **Tic T834**, **Tic T825**, and **Tic T249**. The main differences between those boards are mentioned in **Section 1**.

Each Tic controller is each available in two versions: **with terminal blocks and 0.1″ male headers installed**, as shown in the left pictures above, and **with through-hole connectors included but not soldered in**, as shown in the right pictures above. The versions with connectors installed allow for all of the main features to be used without any additional soldering required as the stepper motor and power leads can be connected to the board via terminal blocks and the signal pins can be connected to the board by 0.1″ connectors, such as **premium jumper wires [https://www.pololu.com/category/65/premium-jumper-wires]**, cables made from **wires with pre-crimped terminals [https://www.pololu.com/category/71/wires-with-pre-crimped-terminals]**, and **servo cables [https://www.pololu.com/category/112/servo-cables]**. The connector-free versions allow for custom installations, such as soldering the included 0.1″ male header pins pointing down for use in a breadboard or soldering wires directly to the board.

- **Tic T500 with terminal blocks and 0.1″ male headers installed [https://www.pololu.com/product/3134]**
- **Tic T500 with through-hole connectors included but not soldered in [https://www.pololu.com/product/3135]**
- **Tic T834 with terminal blocks and 0.1″ male headers installed [https://www.pololu.com/product/3132]**
- **Tic T834 with through-hole connectors included but not soldered in [https://www.pololu.com/product/3133]**
- **Tic T825 with terminal blocks and 0.1″ male headers installed [https://www.pololu.com/product/3130]**
- **Tic T825 with through-hole connectors included but not soldered in**

[https://www.pololu.com/product/3131]

- **Tic T249 with terminal blocks and 0.1″ male headers installed [https://www.pololu.com/product/3138]**
- **Tic T249 with through-hole connectors included but not soldered in [https://www.pololu.com/product/3139]**

The connector-free versions include the following connectors:

- Three **2-pin 3.5mm screw terminal blocks [https://www.pololu.com/product/2444]** – you can combine these by sliding them together and use them with the larger motor and power holes.
- 1×10 **breakaway male header pin strip [https://www.pololu.com/product/965]** – this matches the 10 holes along the side of the board opposite the power and motor connections.
- 1×12 **breakaway male header pin strip [https://www.pololu.com/product/965]** – this can be broken into smaller pieces and used with the other 0.1″ holes on the board as desired.

1.2. Supported operating systems

We support using the Tic Stepper Motor Controller and its configuration software on Windows 7, Windows 8, Windows 10, Linux, and macOS. The Tic's software is not likely to work on Windows 10 IoT Core, which is very different from the normal desktop versions of Windows. The software is **open source [https://github.com/pololu/pololu-tic-software]**, so it could be ported to more platforms.

2. Contacting Pololu

We would be delighted to hear from you about any of your projects and about your experience with the Tic Stepper Motor Controller. You can **contact us** **[https://www.pololu.com/contact]** directly or post on our **forum** **[https://forum.pololu.com/]**. Tell us what we did well, what we could improve, what you would like to see in the future, or anything else you would like to say!

3. Getting started

3.1. Installing Windows drivers and software

To install the drivers and software for the Tic on a computer running Microsoft Windows, follow these steps:

1. Download and install the **Tic Software and Drivers for Windows** **[https://www.pololu.com/file/0J1325/pololu-tic-1.7.0-win.msi]** (9MB msi).
2. During the installation, Windows will ask you if you want to install the drivers. Click “Install” to proceed.
3. After the installation has completed, plug the Tic into your computer via USB. Windows should recognize the Tic and load the drivers that you just installed.
4. Open your Start Menu and search for “Tic”. Select the “Tic Control Center” shortcut (in the Pololu folder) to launch the software.
5. In the upper left corner of the software, where it says “Connected to:”, make sure that it shows something like “#01234567 T825”. This indicates the serial number and model of the Tic that the software has connected to. If it says “Not connected”, see the troubleshooting section below.

The Tic’s USB interface implements Microsoft OS 2.0 Descriptors, so it will work on Windows 8.1 or later without needing any drivers. The driver we provide is needed for earlier versions of Windows.

This Tic software consists of two programs:

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

The **source code for the software** **[https://github.com/pololu/pololu-tic-software]** is available.

USB troubleshooting for Windows

If the Tic software did not connect to the Tic, try opening the “Connected to:” drop-down box to see if there are any entries in the list. If there is an entry, you can select it to connect to it. If there are no entries, then the tips here can help you troubleshoot the Tic’s USB connection on a Windows

computer.

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

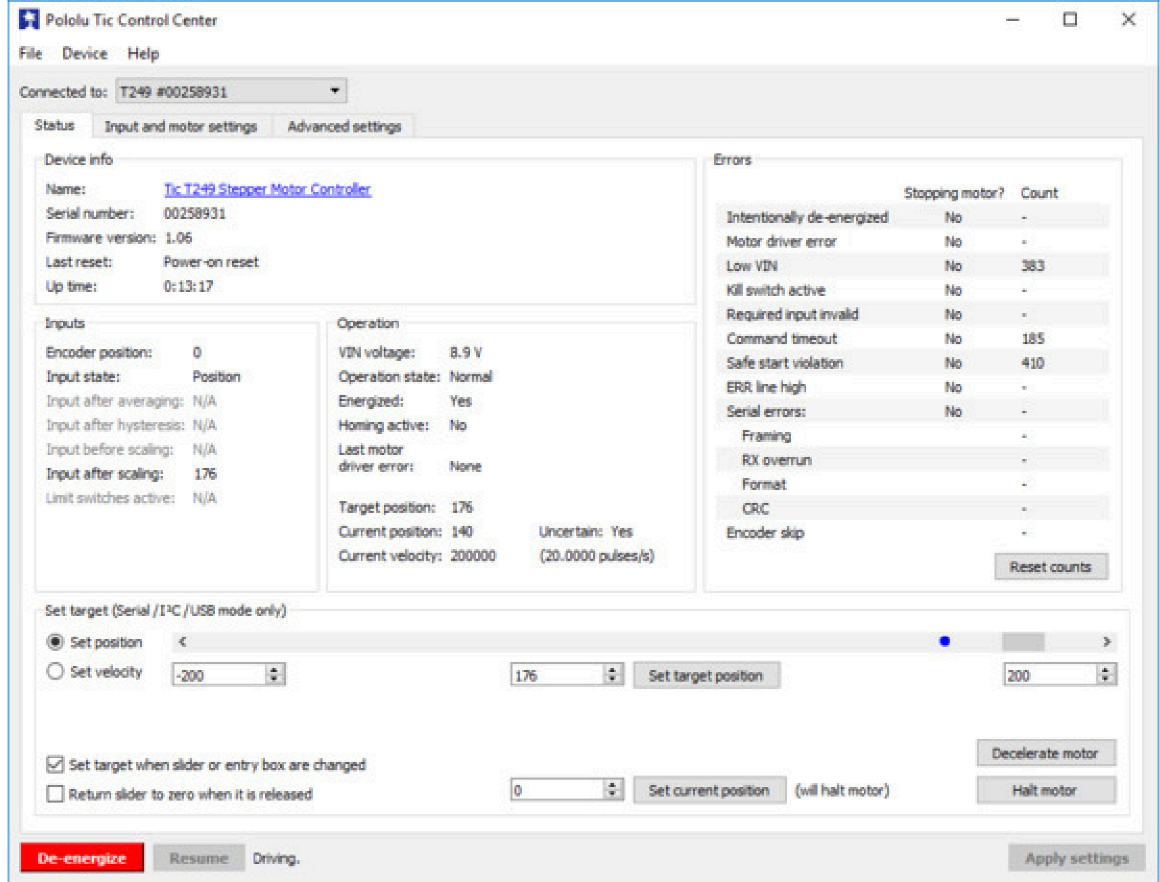
You should look at the LEDs of the Tic. If the LEDs are off, then the Tic is probably not receiving power from the USB port. If the green LED is flashing very briefly once per second, then the Tic is receiving power from USB, but it is not receiving any data. These issues can be caused by using a broken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

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

If that does not help, you should go to your computer's Device Manager and locate all the entries for the Tic. If the Tic's main USB driver is working, you should see an entry in the "Universal Serial Bus devices" category called "Pololu Tic T825". If it has a yellow triangle displayed over its icon, you should double click on the entry to get information about the error that is happening. If you do not see that entry, then you should open the "View" menu and select "Devices by connection". Then expand the entries until you find your computer's USB controllers, hubs, and devices. See if there are any entries in the USB area that disappear when you unplug the Tic. This might give you important information about what is going wrong.

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

 © 2001–2019 Pololu Corporation



The Status tab of the Pololu Tic Control Center.

3.2. Installing Linux software

To install the software for the Tic on a computer running Linux, follow these steps:

1. Download the version for your system from this list:
   - **Tic Software for Linux (x86) [https://www.pololu.com/file/0J1348/pololu-tic-1.7.0-linux-x86.tar.xz]** (8MB xz) — works on 32-bit and 64-bit systems
   - **Tic Software for Linux (Raspberry Pi) [https://www.pololu.com/file/0J1349/pololu-tic-1.7.0-linux-rpi.tar.xz]** (6MB xz) — works on the Raspberry Pi and might work on other ARM Linux systems
2. In a terminal, use `cd` to navigate to the directoy holding the downloaded file. For example, run `cd ~/Downloads` if it was downloaded to the “Downloads” folder in your home directory.
3. Run `tar -xvf pololu-tic-*.tar.xz` to extract the software. If you downloaded multiple versions of the software, you should use an exact file name instead of an asterisk.

4. Run `sudo pololu-tic-*/install.sh` to install the software. You will need to have sudo privilege on your system and you might need to type your password at this point. Look at the output of the script to see if any errors happened.
5. After the installation has completed, plug the Tic into your computer via USB. If you already connected the Tic earlier, unplug it and plug it in again to make sure the newly-installed udev rules are applied.
6. Run `ticcmd --list` to make sure the software can detect the Tic. This command should print the serial number and model of the Tic. If it prints nothing, see the "USB troubleshooting for Linux" section below.
7. Run `ticgui` to start the Tic Control Center.

This Tic software consists of two programs:

- The Pololu Tic Control Center (ticgui) is a graphical user interface (GUI) for configuring the Tic, viewing its status, and controlling it manually. You can open a terminal and type `ticgui` to run it.
- The Pololu Tic Command-line Utility (ticcmd) is a command-line utility that can do most of what the GUI can do, and more. You can open a terminal and type `ticcmd` with no arguments to a see a summary of its options.

No special drivers are needed for the Tic on Linux. Also, the Tic software is statically compiled; it does not depend on any shared libraries.

The **source code for the software [https://github.com/pololu/pololu-tic-software]** is available.

Software installation troubleshooting for Linux

If you do not have sudo privilege or you do not remember your password, you can skip running `install.sh` and just run the Tic programs directly from the directory you extracted them to. For example, try running `pololu-tic/ticgui` after running the `tar` command above. You should also consider moving the software to a more permanent location and adding that location to your PATH as described below.

If you get a "No such file or directory" error while running `./install.sh`, it is possible that your system is missing one of the directories that the install script assumes will be present. Please **contact [https://www.pololu.com/contact]** us to let us know about your system so we can consider supporting it better in the future.

If you get the error "command not found" when you try to run `ticcmd` or `ticgui`, then you should run `echo $PATH` to see what directories are on your PATH, and then make sure one of those directories

contains the Tic executables or symbolic links to them. The installer puts symbolic links in `/usr/local/bin`, so if that directory is not on your PATH, you should run `export PATH=$PATH:/usr/local/bin` to add it. Also, you might want to put that line in your `~/.profile` file so the directory will be on your PATH in future sessions.

If you get the error “cannot execute binary file: Exec format error” when you try to run `ticcmd` or `ticgui`, then it is likely that you downloaded the wrong version of the software from the list above. If all of the listed versions give you this error, you will probably need to compile the software from source by following the instructions in **BUILDING.md** **[https://github.com/pololu/pololu-tic-software/blob/master/BUILDING.md]** in the **source code** **[https://github.com/pololu/pololu-tic-software]**. Please **contact** **[https://www.pololu.com/contact]** us to let us know about your system so we can consider supporting it better in the future.

If the Tic Control Center window is too big to fit on your screen properly, try setting the `TICGUI_COMPACT` environment variable to `Y` before running the software. You can do this by running the command `TICGUI_COMPACT=Y ticgui` in your terminal. You could also add the line `export TICGUI_COMPACT=Y` to your `~/.profile` file to make the change permanent.

If the text in the Tic Control Center window is not visible, make sure that the `.ttf` font file that we ship with the software is in the same directory as the `ticgui` executable.

USB troubleshooting for Linux

If the Tic software cannot connect to your Tic after you plug it into the computer via USB, the tips here can help you troubleshoot the Tic’s USB connection.

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

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

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

If you get a “Permission denied” error when trying to connect to the Tic, you should make sure to copy

the `99-pololu.rules` file into `/etc/udev/rules.d` and then unplug the Tic and plug it back in again. The install script normally takes care of installing that file for you.

If that does not help, you should try running `lsusb` to list the USB devices recognized by your computer. Look for the Pololu vendor ID, which is **1ffb**. You should also try running `dmesg` right after plugging in the Tic to see if there are any messages about it.

3.3. Installing macOS software

To install the software for the Tic on a computer running macOS, follow these steps:

1. Download the **Tic Software for macOS** **[https://www.pololu.com/file/0J1402/pololu-tic-1.7.0-macos.pkg]** (7MB pkg).
2. Double-click on the downloaded file to run it, and follow the instructions.

This Tic software consists of two programs:

- The Pololu Tic Control Center (ticgui) is a graphical user interface (GUI) for configuring the Tic, viewing its status, and controlling it manually. You can run it by double-clicking on "Pololu Tic Stepper Motor Controller" in your "Applications" folder, or by opening a Terminal and typing `ticgui` .
- The Pololu Tic Command-line Utility (ticcmd) is a command-line utility that can do most of what the GUI can do, and more. You can open a terminal and type `ticcmd` with no arguments to see a summary of its options.

The **source code for the software** **[https://github.com/pololu/pololu-tic-software]** is available.

Software installation troubleshooting for macOS

If you get the error "command not found" when you try to run `ticcmd` or `ticgui` , then you should try starting a new Terminal window. The installer for the Tic software adds a file named `99-pololu-tic` in the `/etc/paths.d` directory to make sure the Tic software gets added to your PATH, but the change will not take effect until you open a new Terminal window.

If the Tic Control Center window is too big to fit on your screen properly, try setting the `TICGUI_COMPACT` environment variable to `Y` before running the software. You can do this by running the command `TICGUI_COMPACT=Y ticgui` in the Terminal. You could also add the line `export TICGUI_COMPACT=Y` to your `~/.profile` file to make the change permanent.

USB troubleshooting for macOS

If the Tic software cannot connect to your Tic after you plug it into the computer via USB, the tips here

can help you troubleshoot the Tic’s USB connection.

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

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

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

Another thing to try is to run `dmesg` right after plugging in the Tic to see if there are any messages about it.

3.4. LED feedback

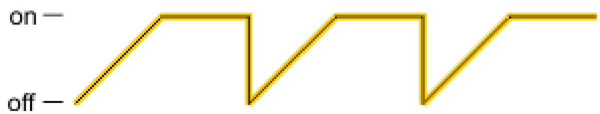
The Tic Stepper Motor Controller has three LEDs to indicate its status.

The **green LED** indicates the USB status of the device. When you connect the Tic to a computer via a USB cable, the green LED will start blinking slowly. The blinking continues until the Tic gets a particular message from the computer that indicates that the Tic is recognized. After the Tic gets that message, the green LED will be on, but it will flicker briefly when there is USB activity. During suspend mode (i.e. when the Tic is only powered from USB and the computer has gone to sleep), the green LED will blink very briefly once per second.

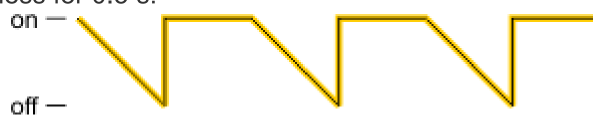
The **red LED** indicates that an error is happening. The red LED is tied to the ERR pin. For more information about error handling, see **Section 5.4**.

The **yellow LED** indicates the status of the stepper motor and also gives some information about what errors are happening, if there are errors:

- If the stepper motor coils are energized (i.e. electrical current is flowing), then the yellow LED will be on solid most of the time.
    - If the motor is moving forward (i.e. the current velocity is positive), the yellow LED will fade in for 0.5 s and then stay on at full brightness for 0.5 s:



- If the motor is moving in reverse, the yellow LED will fade out for 0.5 and then turn on at full brightness for 0.5 s:



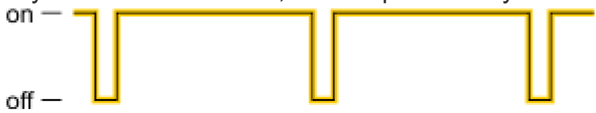
- If the motor is energized, but not moving, and there are no errors, the yellow LED will turn on at full brightness:

on

off

- If the motor is energized and there is any error happening other than a safe start violation, the yellow LED will stay on most of the time, but will periodically blink off twice:

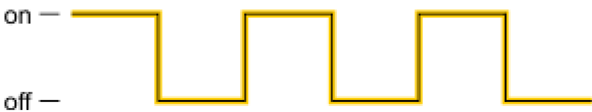
- If the motor is energized and a safe start violation is only error happening, then the yellow LED will stay on most of the time, but will periodically blink off once:



- If the stepper motor coils are de-energized, then the yellow LED will be off most of the time.
  - If there is a motor driver error (i.e. from an over-current or over-temperature condition), the yellow LED will blink 8 times per second:

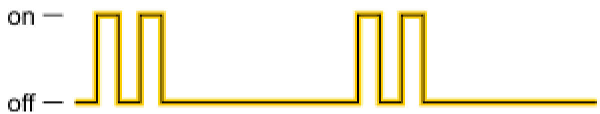
This blinking will look very similar to the abnormal startup blinking described below, so if you see it, then it is best to look at the Tic's status using the Tic Control Center software to figure out what is happening.

  - If there is no motor power supplied to VIN, or the voltage is too low, the yellow LED will blink slowly once per second (on for 0.5 s, off for 0.5 s):

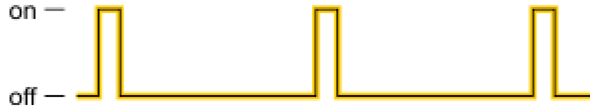


- Otherwise, if the motor has been intentionally de-energized by receiving the De-energize command over USB, serial, or I²C, then the yellow LED will be off most of the time, but periodically blink three times:

- Otherwise, if the motor is de-energized and any error is happening other than a safe start violation, then the yellow LED will be off most of the time, but periodically blink two times:



- If the motor is de-energized and the only error is a safe start violation, then the yellow LED will be off most of the time, but periodically blink once:



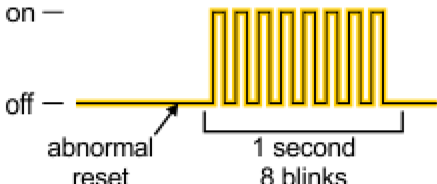
Note that the Tic does not read any feedback from the motor so it cannot tell if the stepper motor is moving, or if it is even connected. The LED blinking patterns above are based on the signals that the Tic is sending to the on-board stepper motor driver.

To ensure that the Tic shows a complete blinking pattern instead of switching quickly between two or more patterns, the yellow LED blinking pattern is only updated after it has completed. So if you make a change to your system, you might have to wait for one or two seconds to see the LEDs respond.

The information expressed by the Tic's LEDs can also be seen by connecting the Tic to a computer via USB, running the Tic Control Center, and looking in the Status tab.

Startup blinking

When the Tic starts running, it tries to detect if it was reset by some special condition. If the Tic experiences a brown-out reset, watchdog timer reset, software reset, stack overflow, or stack underflow, the Tic will blink its yellow LED eight times over a one second period while the red LED is on at startup. While it is doing this blinking, the Tic will not accept any commands, read any inputs, or energize the stepper motor. You can see the cause of the last reset in the Tic Control Center software's "Last reset" field. This startup blinking was added in firmware version 1.02.



Bootloader mode

In bootloader mode, which is used for updating the firmware of the Tic and should only rarely be needed, the LEDs behave differently. The green LED still indicates the USB status, but it is different: after the bootloader gets a particular message from the computer that indicates that the bootloader is recognized, the green LED will start doing a double blinking pattern every 1.4 seconds. The yellow LED will usually be on solid, but it will blink quickly whenever a USB command is received. The red LED will be on if and only if there is no firmware currently loaded on the device.

© 2001–2019 Pololu Corporation

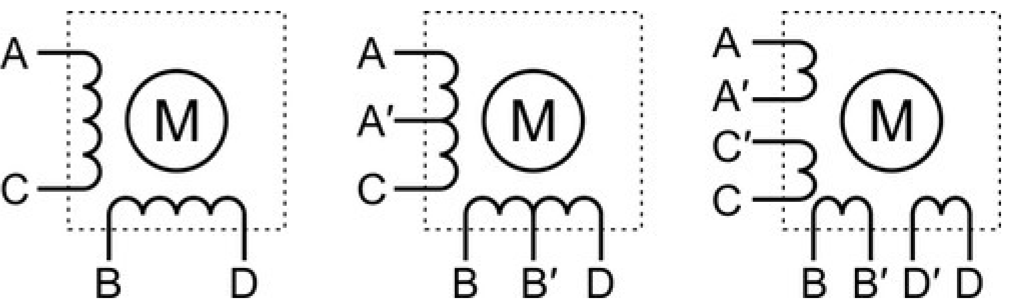
4. Setting up the controller

4.1. Choosing the power supply, Tic, and stepper motor

The information in this section can help you select a **power supply** **[https://www.pololu.com/category/84/regulators-and-power-supplies]**, a Tic controller, and a **stepper motor** **[https://www.pololu.com/category/87/stepper-motors]** that will work well together.

Stepper motor configurations

There are different types of stepper motors. The Tic is designed to work with two-phase stepper motors that can be connected in a bipolar configuration. The following diagram shows examples of such stepper motors:



The Tic works with two-phase stepper motors that can be controlled in a bipolar configuration.

Your stepper motor should have one of the configurations shown above, or else it might not work with the Tic. The **next section** **[https://www.pololu.com/docs/0J71/4.2]** explains how to connect these stepper motors to your Tic.

Voltage and current ratings

When selecting your power supply, Tic controller, and stepper motor, you must consider the voltage and current ratings of each.

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 Tic** is the range of voltages from which it can be powered. The operating voltages of the different Tic controllers are shown in the table below. The Tic requires a DC power supply.

The **continuous current per phase of a Tic** is the maximum amount of current that the Tic can continuously provide to each phase of the stepper motor. The continuous current per phase of the different Tic controllers are shown in the table below.

	Tic T500	Tic T834	Tic T825	Tic T249
Operating voltage range:	4.5 V to 35 V	2.5 V to 10.8 V	8.5 V to 45 V	10 V to 47 V
Continuous current per phase:	1.5 A	1.5 A	1.5 A	1.8 A

Note: While the Tic T500 can operate down to 4.5 V, power supply voltages under 5.5 V could cause a drop in the logic voltage of the board, potentially down to around 4 V when the power supply is 4.5 V. This logic voltage drop causes the Tic's VIN voltage measurements to become inaccurate (too high). If you are connecting analog voltages powered from an external source to the Tic, the lower logic voltage will cause the Tic's analog readings to rise and possibly require recalibration. The logic voltage drop should not affect a potentiometer that is connected to the Tic in the standard way using the GND, SDA/AN, and SCL pins, since the voltage on the potentiometer output will drop by the same percentage as the Tic's logic voltage.

The **rated current of a stepper motor** is the maximum amount of current that the stepper motor was designed to have flowing through each phase, and this is typically the current required to achieve the stepper motor's published performance specifications.

The **rated voltage of a stepper motor** is how much voltage needs to be applied to a coil of the stepper motor to get the rated current to flow through it. **Ohm's law [https://en.wikipedia.org/wiki/Ohm%27s_law]** provides the simple relationship between the rated voltage and the rated current: the rated voltage is equal to the rated current multiplied by the coil resistance.

These are the main constraints you should keep in mind when selecting your power supply, Tic controller, and stepper motor:

1. The voltage of your power supply should be greater than or equal to the rated voltage of your stepper motor. Otherwise, the motor will not receive its full rated current and you will not get the full performance that the motor is capable of. It is OK for the power supply voltage to be higher than the rated voltage of the motor because the Tic has active current limiting. (It rapidly switches the power to the motor on and off while measuring the current to make sure it does not go too high.)

2. A higher power supply voltage is usually desirable since it allows higher speed and torque. However, if the power supply voltage is extremely high compared to the stepper motor’s rated voltage and you want to use microstepping, you might experience skipped steps.
3. The voltage of your power supply should be within the operating voltage range of the Tic. Otherwise, the Tic could malfunction or (in the case of high voltages) be damaged.
4. The continuous current per phase of the Tic should be greater than or equal to the rated current of the stepper motor. Otherwise, the Tic will not be able to deliver the full rated current to the motor and you will not get the full performance that your motor is capable of. (However, if you are using USB, serial, or I²C to control the Tic, you might be able to get better performance out of the Tic by dynamically increasing the current limit above the Tic’s continuous current rating whenever you move the motor, and reducing it while the motor is holding position, thus maintaining a low average current.) If the motor’s rated current is substantially more than the Tic’s current, then it is possible that the Tic will not be able to move the motor at all.
5. We generally recommend you choose a power supply with a current limit that is at least at least twice the current limit you are planning to use on the Tic as that amount of current should always be safely beyond what the Tic will draw. The current limit you configure on the Tic should generally not exceed the stepper motor’s rated current and should not exceed the continuous current per phase of the Tic. So if you take the smaller of those two currents, and then multiply that current by two, and get a power supply that can provide at least the much current, you can be sure that the power supply’s current will not be the limiting factor in your application. However, please note that you can typically get by with less power supply current than this, especially if your supply voltage is higher than the rated voltage of your stepper motor. In this situation, the Tic’s current control acts as a step-down converter, meaning that a small amount of current from the power supply at a higher voltage can generate a larger amount of current going through the coils at a lower voltage. Also, the Tic T500, Tic T825, and Tic T834 never actually drive both coils at the configured current limit at the same time. The total current going through the coils is maximized in the four full-step positions, where the Tic will be sending 71% of the current limit through each coil, for a total current of 142% of the current limit. (The Tic T249 has two step modes where it will send the full configured current into both coils at the same time.) If you want to know the maximum current draw from your power supply, you can measure this with a multimeter while the stepper motor is energized in full-step mode and not stepping. If your system draws too much current, your power supply might shut down, overheat, produce a lower voltage, and/or be damaged.

© 2001–2019 Pololu Corporation

It is worth noting again that since the Tic actively limits current through the motor coils, you can safely use power supplies with voltages above the rated voltage of the stepper motor as long as you set the current limit to not exceed the stepper motor's rated current. This means that the Tic T825 and Tic T249 can be used with stepper motors that have rated voltages below 8.5 V.

4.2. Connecting the stepper motor and power supply

The information in this section can help you connect your stepper motor and power supply to the Tic.

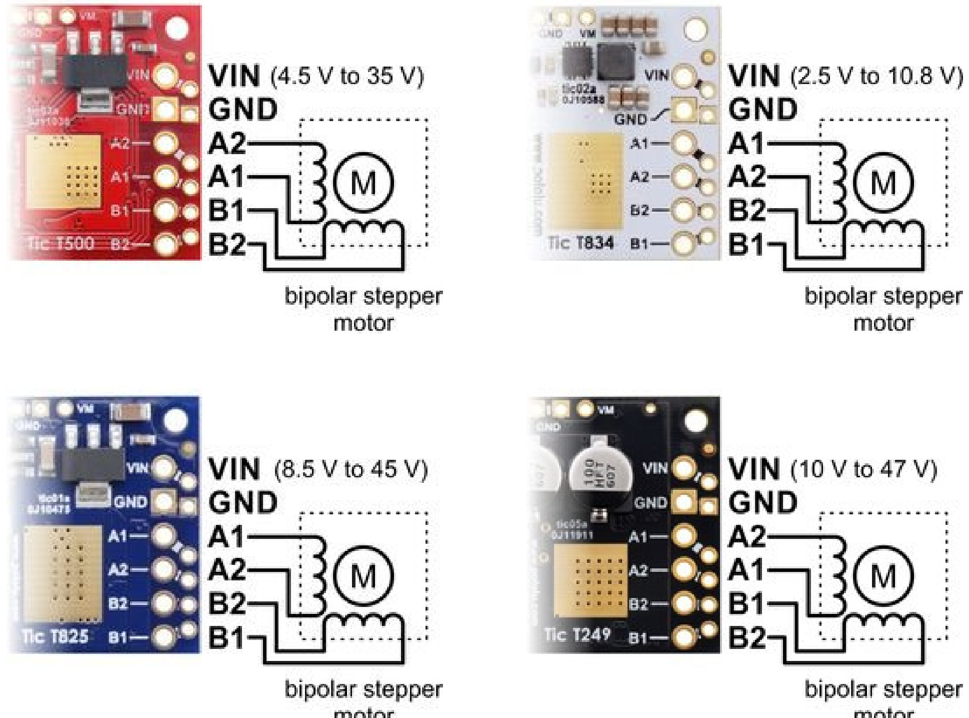
To avoid damage or injury, please read these safety warnings carefully:

Warning: This product is not designed to or certified for any particular high-voltage safety standard. Working with voltages above 30 V can be extremely dangerous and should only be attempted by qualified individuals with appropriate equipment and protective gear.

Warning: Connecting or disconnecting a stepper motor while the Tic's motor power supply (VIN) is powered can destroy the motor driver. (More generally, rewiring anything while it is powered is asking for trouble.)

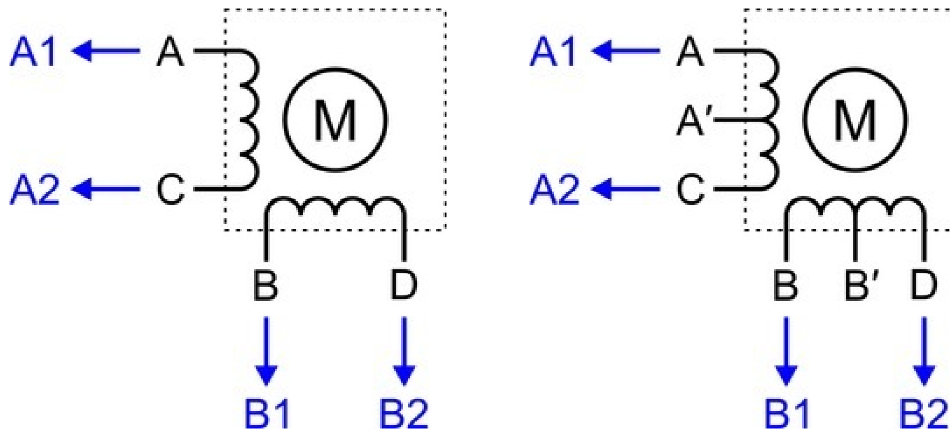
Warning: This product can get hot enough to burn you long before the chips overheat. Take care when handling this product and other components connected to it.

Before connecting anything to the Tic, we recommend running the Tic Control Center software to make sure it can connect to the Tic over USB. This way you can ensure that the Tic is functioning before you spend time soldering connectors or connecting other electronics, and if something goes wrong, you will have a better idea of what caused the problem.



Connecting a bipolar stepper motor with four or six leads

Bipolar stepper motors commonly have four or six leads. These two-phase stepper motors have one coil per phase, with one lead connected to each end of each coil. Versions with six leads also provide access to the centers of the two coils so that the motor can optionally be controlled by a unipolar driver. When controlling a six-lead stepper motor with a bipolar driver like the Tic, only the ends of the coils are used, and the two center tap leads should be left disconnected.



Connecting a two-phase bipolar stepper motor with four (4) leads to the Tic.

Connecting a two-phase unipolar/bipolar stepper motor with six (6) leads to the Tic. The two center taps are left disconnected.

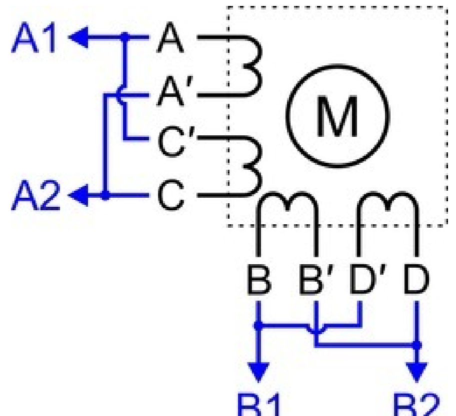
© 2001–2019 Pololu Corporation

Swapping A1 with A2 or B1 with B2 in the above diagrams just reverses the direction of the motor. Swapping both will leave the direction unchanged.

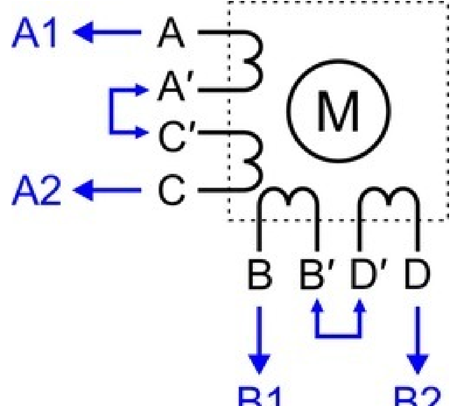
Connecting a bipolar stepper motor with eight leads

Unlike four- and six-lead stepper motors, which have a single coil per phase, eight-lead unipolar/bipolar stepper motor have two coils per phase and give you access to all of the coil leads. You have the option of using the two coils for each phase in parallel or in series.

When using them in parallel, you decrease coil inductance, which can lead to increased performance if you have the ability to deliver more current. However, since the Tic actively limits the output current per phase, you will only get half the phase current flowing through each of the two parallel coils. When using the phase coils in series, it's like having a single coil per phase (like the motor types discussed above). We generally recommend using a series connection. The following diagram shows how to connect such a stepper motor to the Tic with each pair of phase coils in parallel (left) or series (right):



Connecting a two-phase unipolar/bipolar stepper motor with eight (8) leads to the Tic with the phase coils in parallel.



Connecting a two-phase unipolar/bipolar stepper motor with eight (8) leads to the Tic with the phase coils in series.

Power supply

To connect your power supply to the Tic, connect the negative or ground terminal of your power supply to the Tic's GND pin on the high current side of the board (next to motor output A1). Then, connect your power supply's positive terminal to the VIN pin next to that.

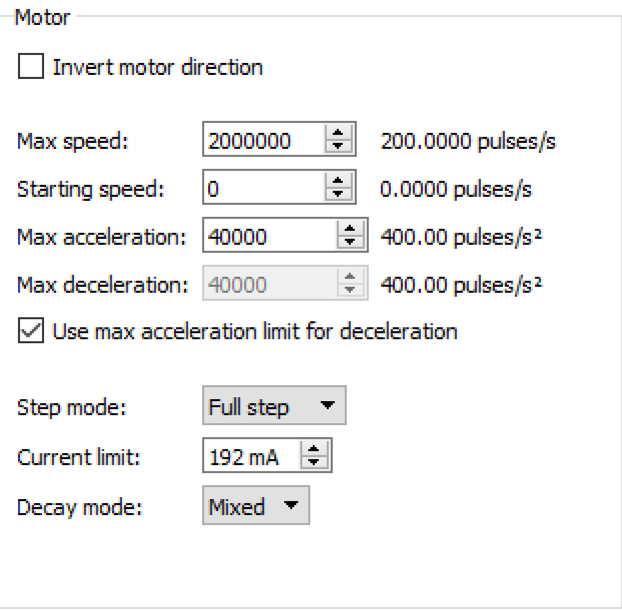
4.3. Configuring and testing the stepper motor

This section explains how to configure and test your motor over USB using the Tic Control Center software. It is a good idea to test the motor over USB like this to make sure that the motor is working and that you can get the desired performance out of it before you connect a different kind of input to

the Tic and try to use that to control the motor.

If you have changed any of the settings of your Tic, you should probably reset the Tic to its default settings by opening the "Device" menu and selecting "Restore default settings". Then, go to the "Input and motor settings" tab and make sure the "Control mode" is set to "Serial / I²C / USB" (the default).

The Tic's motor settings can be found in the "Motor" box. This screenshot shows the default settings for the Tic T825:



The default motor settings for the Tic T825.

The different Tic models have different available step modes, current limits, and decay modes. The Tic T249 has four special settings for configuring its Active Gain Control (AGC) feature, as described in the **Section 6**.

Setting the current limit

Assuming that you are not limited by the Tic or your power supply, we recommend setting the current limit of the Tic to the rated current of your motor. You should make sure the current limit is not higher than what the Tic can deliver continuously (without a heatsink or forced air flow: 1500 mA for the T825, T834, or T500, and 1800 mA for the Tic T249), and make sure the current limit is not higher than half of the rated current of your motor power supply (though this is not always necessary and a higher current limit could work, as explained in **Section 4.1**). You should also make sure that the Tic's configured

current limit never exceeds the rated current of your stepper motor.

The current limit is specified in the Tic Control Center in units of milliamps (mA), which are one thousandth of an amp (ampere). So if you want to set your current limit to 0.9 A, you should enter "900" in the "Current limit" field. Note that the current limit can only be set to certain specific values. After you type in a current limit, the control center will use the closest valid setting that is less than or equal to the current limit you typed. You can use the up and down arrows to browse through the valid current limit settings. The different Tic models have different sets of allowed current limits.

Testing the motor for the first time

After setting the current limit, click "Apply settings". There should be a message at the bottom of the window that says "Motor de-energized because of safe start violation. Press Resume to start." Click the green "Resume" button to energize the stepper motor. If all goes well, the current you have selected will start flowing through the coils, and the message at the bottom of the screen should change to "Driving".

If your power supply cannot supply enough current, its voltage might dip when you click the "Resume" button. The Tic will detect that VIN has dropped too low (7.0 V for the Tic T825, 2.1 V for the Tic T834, 3.0 V for the Tic T500, 5.5 V for the Tic T249), report a "Low VIN" error, and de-energize the motor. If this is happening in your system, what you will see is that the Tic drives the motor briefly and then switches back to the previous state, where the motor is de-energized because of a safe start violation. You can also look in the "Status" tab to see if the "Low VIN" error has occurred: the count next to that error would be non-zero, and increase every time you click "Resume". If your power supply voltage is around 2.1 V to 2.3 V or drops to that level when the motor is energized, the Tic T834 might report a "Motor driver error" (caused by the DRV8834's under-voltage lockout fault) without reporting a "Low VIN" error. An inadequate power supply can also cause other problems, such as disrupting the USB communication or making the Tic reset. If your system is having problems like this, you should try getting a better power supply or lowering the current limit to address these issues before continuing.

Next, go to the "Status" tab and use the "Set target" interface at the bottom of that tab to command your motor to go to different target positions. If the "Set target when slider or entry box are changed" checkbox is checked, you can move the stepper motor by just dragging the scrollbar around. You should make sure that your stepper motor can turn in both directions. If the stepper motor is not moving correctly, you should turn off your motor power, check all of your connections (and soldering joints, if applicable), and try again.

Checking the heat

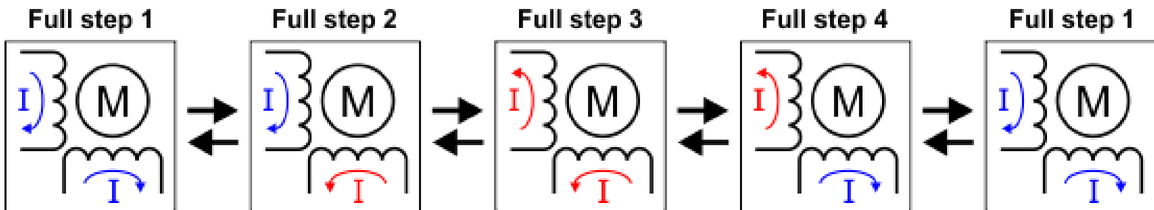
After you have gotten your motor to move, you might want to let the motor hold position for a while to see how hot the motor and the Tic get. Unlike a DC motor, stepper motors consume power and generate heat while they are not moving. After your system heats up and reaches a steady state, if

the motor or the Tic are hotter than you would like them to be, you might consider lowering the current limit.

Warning: This product can get hot enough to burn you long before the chips overheat. Take care when handling this product and other components connected to it.

Setting the step mode

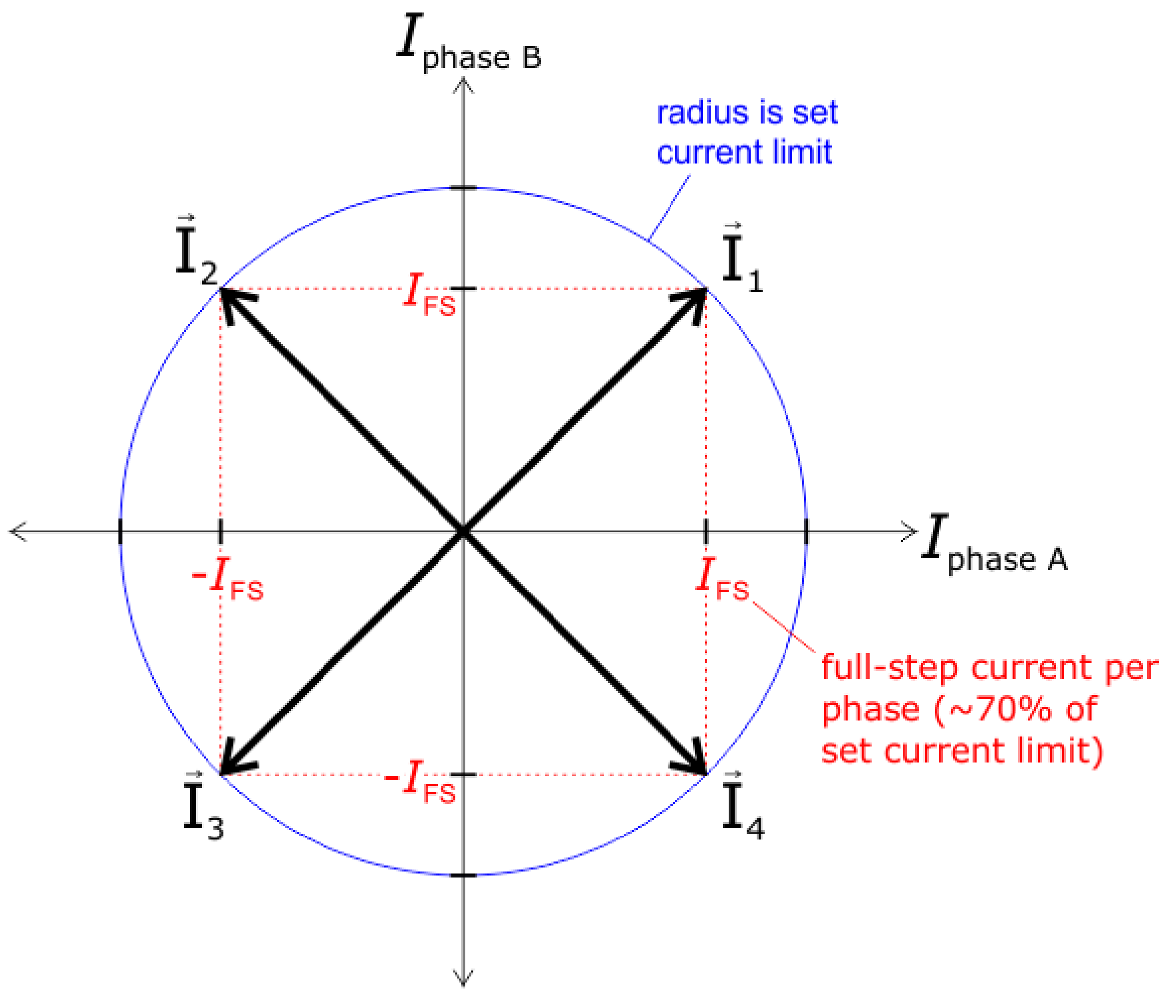
The "Step mode" setting controls the microstepping resolution of the Tic. In the "Full step" and "Full step 100%" modes, the same amount of current is always flowing through both coils, and every time the Tic takes a step, it will reverse the current in one of the coils. There are only four possible coil current states in full-stepping mode, as shown in the diagram below, and if your motor's documentation says that it has 200 steps per revolution, that means that it takes 200 of these full steps to rotate 360 degrees.



Allowed coil current transitions in full step mode. Arrows to the right correspond to one motor rotation direction and arrows to the left correspond to the other.

Another way to visualize this is with a graph of the coil currents for each of the four full steps, with one axis representing the phase (or coil) A current and the other axis representing the phase B current. For "Full step" mode, where each coil always has approximately 70% of the configured current limit, that graph looks like this:

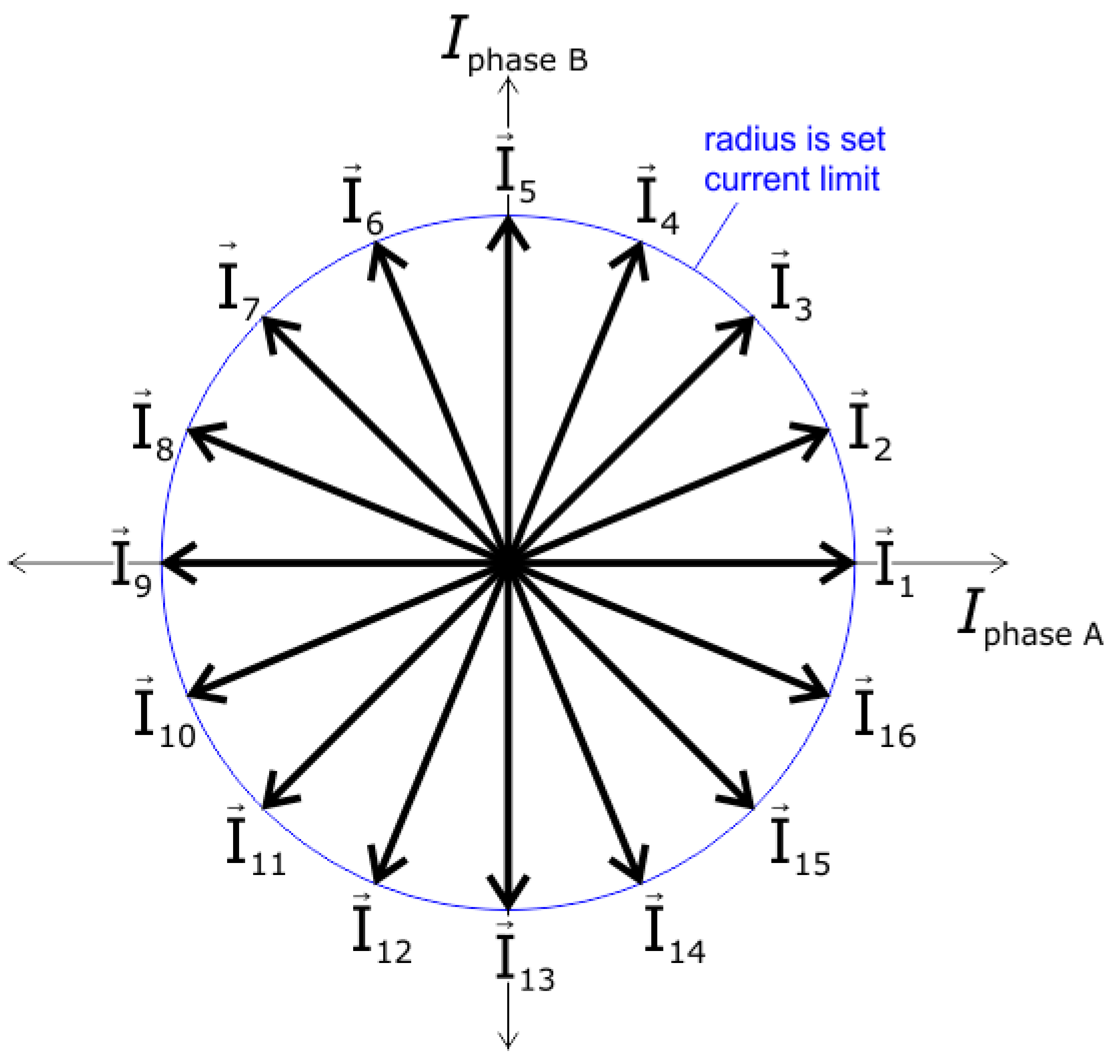
© 2001–2019 Pololu Corporation



Coil currents for the four (4) steps that make up full-step mode (I1-4).

In the other available step modes, the Tic uses microsteps instead of full steps to generate magnetic fields that point to places between the full steps. Each microstep corresponds to 1/2, 1/4, 1/8, 1/16, or 1/32 of a full step, depending on what step mode you choose. For example, if you choose 1/32, then it will take 32 microsteps to move the same distance as one full step, and a motor with 200 steps per revolution will require 6400 microsteps to turn 360 degrees. The following graph shows the coil currents for each of the microsteps in 1/4-step mode:

 © 2001–2019 Pololu Corporation



Coil currents for the 16 microsteps that make up 1/4-step mode ($I_{1\text{-}16}$).

In the above graph, currents I_3, I_7, I_{11}, and I_{15} match the four full-step currents, where the magnitude of the current through both coils is equal. All other steps point between these full steps by setting different current limits for the two coils. The most extreme example of this occurs on steps I_1, I_5, I_9, and I_{13}, where the current through one coil is equal to the full current limit setting on the Tic while the current through the other coil is zero.

The Tic T825 and Tic T834 support full, 1/2, 1/4, 1/8, 1/16, and 1/32 step modes. The Tic T500 only supports full, 1/2, 1/4, and 1/8 step modes.

The Tic T249 supports "Full step 100%", "1/2 step", "1/2 step 100%", "1/4 step", "1/8 step", "1/16 step", and "1/32 step" modes. In the "Full step 100%" and "1/2 step 100%" modes, any non-zero coil current is 100% of the rated current instead of 70%. These modes are sometimes called "non-circular".

The Tic's speed, velocity, acceleration, and deceleration numbers are all denominated in microsteps, which are also called pulses. Therefore, if you change the step mode, you might have to change those other settings to account for the change. For example, the default maximum speed for the Tic is 200 pulses per second. If you change the step mode from full step to half step, you would have to change the speed to 400 pulses per second to maintain the same angular rate of change. Since the step mode affects those other parameters, it is a good idea to set it first.

Setting the decay mode (T825 and T834 only)

The Tic T825 and Tic T834 have a decay mode setting that affects how fast current through the motor coils decays during each step. The Tic T825 has three decay modes: slow, mixed (default), and fast. The Tic T834 has five decay modes: slow, mixed 25%, mixed 50% (default), mixed 75%, and fast. The decay mode matters most when microstepping is used. Which decay mode is most appropriate depends on many factors specific to a particular stepper motor system, including the motor's resistance and inductance, the supplied motor voltage, and the desired speed. Generally, using slow decay generates less electrical and audible noise, but it can result in missed microsteps when the coil current is decreasing. Fast decay is much noisier both electrically and audibly, but it creates more evenly sized microsteps. Mixed decay is a combination of both fast and slow decay that tries to minimize noise while keeping microsteps as even as possible. For more information about the decay modes and current control methods of the Tic's stepper motor driver, refer to the **DRV8825 datasheet** **[https://www.pololu.com/file/0J590/drv8825.pdf]** (1MB pdf) for the Tic T825 or the **DRV8834 datasheet** **[https://www.pololu.com/file/0J617/drv8834.pdf]** (2MB pdf) for the Tic T834.

The decay mode of the Tic T500 is not configurable. The Tic T500 features automatic decay mode selection, using internal current sensing to automatically adjust the decay mode as necessary to provide the smoothest current waveform. For more information, see the **MP6500 datasheet** **[https://www.pololu.com/file/0J1447/MP6500_r1.0.pdf]** (1MB pdf).

The decay mode of the Tic T249 is not configurable. The Tic T249 features Toshiba's Advanced Dynamic Mixed Decay (ADMD) technology, which dynamically switches between slow and fast decay modes based on the actual motor current, providing higher efficiency and smoother steps at high speed than you get with traditional timing-based mixed decay. For more information, see the **TB67S249FTG datasheet** **[https://www.pololu.com/file/0J1523/TB67S249FTG_datasheet_en_20170818.pdf]** (533k pdf).

Configuring Active Gain Control (T249 only)

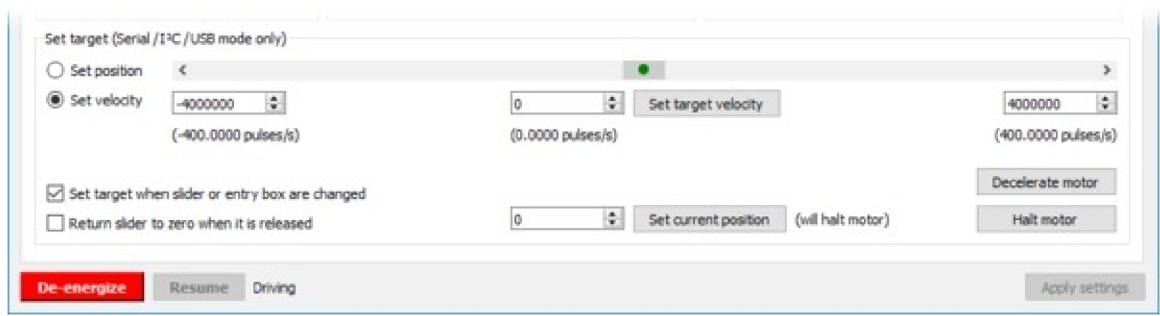
The TB67S249FTG IC on the Tic T249 supports Toshiba's Active Gain Control (AGC) feature, which automatically reduces the stepper motor current below the set limit based on the actual load on the

motor, allowing for reduced unnecessary heat generation and higher peak power when the motor actually needs it. The AGC feature is disabled by default, but you can enable it and configure it using the AGC settings in the "Input and motor settings" tab. For more information about these settings, see **Section 6**.

Setting the movement parameters

After you have set the motor's step mode, current limit, decay mode, and AGC settings (if applicable), you should set its maximum speed and maximum acceleration.

The Tic represents speeds (non-negative values indicating the magnitude of a velocity), velocities (signed values indicating speed and direction), and speed limits in units of pulses (microsteps) per 10,000 (ten thousand) seconds. The Tic can send up to 50,000 pulses (microsteps) per second, so the maximum allowed speed setting is 500,000,000. However, your motor might not be capable of moving that fast. If you want to get the maximum speed possible out of your motor, you might have to do some tests to see how fast it can go. To do this, set the max speed to 500,000,000 in the "Input and motor settings" and click "Apply settings". Then go to the "Set target" box in the "Status" tab, select "Set velocity", and enter smaller numbers in the boxes at the ends of the scroll bar that determine its range. For example, try entering -4,000,000 and 4,000,000, which would mean the scrollbar can set target velocities between −400 pulses per second and +400 pulses per second, as shown below.



The "Set target" box in the Tic Control Center, with its range set to plus or minus 400 pulses per second.

Try slowly dragging the scrollbar to both ends of its range. If your motor is able to reach the desired speeds without pausing or skipping, you can increase the range of the scrollbar and try again. By experimenting with the velocity scrollbar, you should be able to get an idea of what your motor can do. Once you have done that, go back to the "Input and motor settings" tab and set the "Max speed" appropriately.

You will probably have to adjust the "Max acceleration" parameter too. The Tic represents acceleration and deceleration limits in units of pulses per second per 100 seconds. The acceleration and deceleration limits specify how much the speed (in units of 10,000 pulses per second) is allowed to rise or fall in one hundredth of a second (0.01 s or 10 ms). To set the acceleration limit, you might

consider how much time you want the Tic to spend accelerating from rest to full speed. If you want it to take one second, then set the maximum acceleration to be one hundredth of the maximum speed.

Unlike a DC motor, which will accelerate on its own up to some max speed when a voltage is applied, step rates must be gradually increased by the controller if you want to achieve high maximum speeds. If you just jump abruptly to a high step rate, the inertia of the stationary rotor will prevent it from being able to keep up with the rotating magnetic field and it will get left behind; the result of this is that it will just sit there or vibrate in place (or possibly even start moving backward). The Tic's max acceleration parameter limits how quickly the step rates will increase, which if set correctly, will give the rotor time to keep up with the magnetic field as it spins faster and faster (up to some maximum speed that is ultimately a function of your specific stepper motor as well as the current limit setting, the supply voltage, and the load on the stepper motor output).

By default, the Tic's deceleration limit is the same as its acceleration limit. If you want the deceleration rate to be different, you can uncheck the "Use max acceleration limit for deceleration" box.

The "Starting speed" parameter specifies a speed below which deceleration and acceleration limits are not respected. For example, if you set the starting speed to 1000000 (100 pulses per second), then the Tic will be able to instantly change from any velocity in the range of −1000000 to +1000000 to any other velocity in that range. Setting the starting speed might allow you to make your system faster since it will not waste time accelerating or decelerating through low speeds where it is not needed.

4.4. Setting up USB control

This section explains how to control the Tic over USB.

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor. You should leave your Tic's control mode set to "Serial / I²C / USB" (the default). That section also shows how to control the Tic over USB using the Tic Control Center software. If the Tic Control Center's graphical user interface is good enough for you, you do not need to set up anything else and can skip the rest of this section.

Another option for controlling the Tic over USB is to use the Tic Command-line Utility, `ticcmd`. You can either run the utility directly by typing commands in your command prompt (shell), or you can write your own software that runs it.

To try out `ticcmd`, you should open a new command prompt and run `ticcmd` without any arguments. This causes `ticcmd` to print out a help screen listing all the options it supports. You can combine multiple options in one invocation.

If your command prompt prints out a message indicating that `ticcmd` is not found or not recognized, make sure you have installed the Tic software properly as described earlier in this guide. Also, make sure that the directory containing the `ticcmd` executable is in your PATH environment variable, and try starting a new command prompt or terminal after installing the software.

To set the target position or velocity of the Tic, try running these commands:

```
ticcmd --exit-safe-start --position 400
ticcmd --exit-safe-start --velocity 2000000
```

Note: The `--position` option can be abbreviated as `-p`, and `--velocity` can be abbreviated as `-y`.

If the commands above do not produce movement, you should run `ticcmd --status` to print out the errors that are currently stopping the motor. This might tell you what is going wrong.

On Microsoft Windows, only one device can access the Tic's USB interface at a time, so you will need to close the Tic Control Center software before running `ticcmd`.

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can run `ticcmd --reset-command-timeout` every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

To get the current status of the Tic, try running these commands, which give different levels of detail:

```
ticcmd --status
ticcmd --status --full
```

The output of these commands is designed to be compatible with the YAML format, so if you are writing a computer program that needs to get some information from the Tic, you can parse the output with a YAML parser in the language of your choice.

If you have multiple Tics connected to the computer, you will need to use the `-d` option to specify the serial number of the device you want to use. For example, to get the status of the Tic with serial number 12345678, run `ticcmd -d 12345678 --status`. You can run `ticcmd --list` to get the serial numbers (they are listed in the first column). If you omit the `-d` option, `ticcmd` will print: "Error: There are multiple qualifying devices connected to this computer. Use the -d option to specify which device

you want to use, or disconnect the others."

For more details about the commands you can send to the Tic over USB, see **Section 8**.

If you want to write your own software to control the Tic instead of just using `ticcmd` or the Tic Control Center, see **Section 12**.

4.5. Setting up serial control

This section explains what kind of serial interface the Tic has and how to connect a microcontroller or other TTL serial device to it so that you can send commands to control the Tic. The **Tic Stepper Motor Controller library for Arduino** **[https://github.com/pololu/tic-arduino]** makes it particularly easy to control the Tic from an Arduino or Arduino-compatible board such as an **A-Star 32U4** **[https://www.pololu.com/a-star]**.

About the serial interface

The **RX** and **TX** pins of the Tic provide its serial interface. The Tic's RX pin is an input, and its TX pin is an output. Each pin has an integrated 100 kΩ resistor pulling it up to 5 V and a 220 Ω or 470 Ω series resistor protecting it from short circuits.

The serial interface uses non-inverted TTL logic levels: a level of 0 V corresponds to a value of 0, and a level of 5 V corresponds to a value of 1. The input signal on the RX pin must reach at least 4 V to be guaranteed to be read as high, but 3.3 V signals on RX typically work anyway.

The serial interface is *asynchronous*, meaning that the sender and receiver each independently time the serial bits. The sender and receiver must be configured to use the same *baud rate*, which is typically expressed in units of bits per second. The data format is 8 data bits, one stop bit, with no parity, which is often expressed as **8-N-1**. The diagram below depicts a typical serial byte:

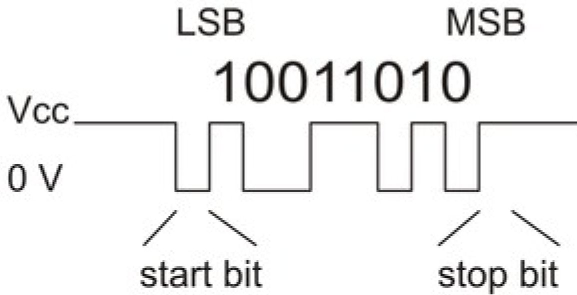


Diagram of a non-inverted TTL serial byte.

The serial lines are high by default. The beginning of a transmitted byte is signaled with a single low *start bit*, followed by the bits of byte, least-significant bit (LSB) first. The byte is terminated by a *stop*

© 2001–2019 Pololu Corporation

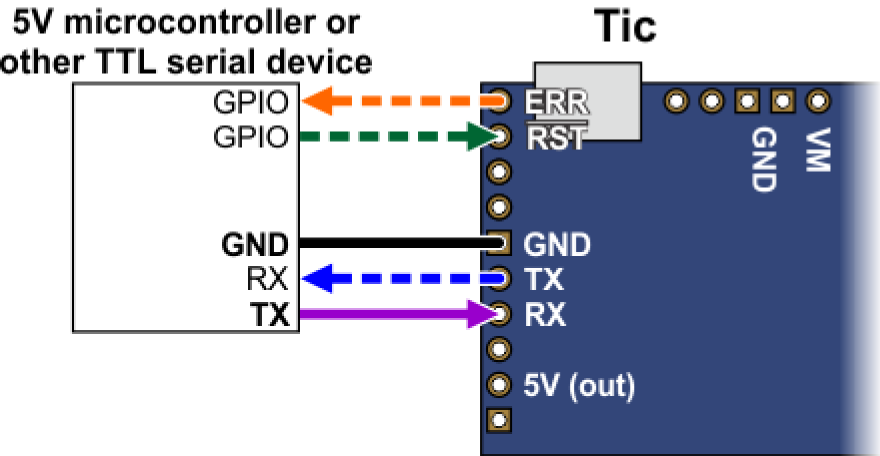
bit, which is the line going high for at least one bit time.

Connecting a serial device to one Tic

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor. You should leave your Tic's control mode set to "Serial / I²C / USB" (the default), and you should also set your desired baud rate.

Next, connect your serial device's GND (ground) pin to one of the Tic's GND pins.

If your serial device operates at 5 V, you can directly connect the device's TX line to the Tic's RX line and connect the Tic's TX line to the device's RX line. The connection to the Tic's TX line is only needed if you want to read data back from the Tic. These connections, and some other optional connections, are shown in the diagram below:



If your serial device operates at 3.3 V, then you might need additional circuitry to shift the voltage levels. You can try connecting the device's TX line directly to the Tic's RX line; this will usually work, but the input signal on the RX pin must reach at least 4 V to be guaranteed to be read as high. If you want to read data from the Tic, you will need to consider how to connect the Tic's TX line to your device's RX line. If your device's RX line is 5V tolerant, meaning that it can accept a 5 V output being applied directly to it, then you should be able to connect the Tic's TX line directly to your device's RX line. If your device's RX line is not 5V tolerant, you will need to a use a level shifter—a separate board or chip that can convert 5 V signals down to 3.3 V. A voltage divider made with two resistors would work too.

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

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

If you are using an Arduino or Arduino-compatible board, you should now try running the SerialSpeedControl example that comes with the Tic Arduino library. The library's **README** **[https://github.com/pololu/tic-arduino]** has information about how to get started and which pins of the Arduino to use. If you are using a different kind of microcontroller board, you will need to find or write code to control the Tic on your platform. If you are writing your own code, we recommend that you first learn how to send and receive serial bytes on your platform, and then use the SerialSpeedControl example as a reference. You should also refer to the sections in this guide about the Tic's commands (**Section 8**) and serial protocol (**Section 9**).

If your connections and code are OK, you should now see the Tic's stepper motor moving back and forth. If the stepper motor is not moving, you should check all of your connections and solder joints (if applicable). You should make sure that the Tic and your device are configured to use the same baud rate. The Tic uses 9600 bits per second by default. You should also check the "Status" tab of the Tic Control Center to see if any errors are being reported.

The SerialSpeedControl example only writes data to the Tic, so it does not test your connection to the Tic's TX line. If you want to read data from the Tic, you should now try the SerialPositionControl example, which reads the current position of the stepper motor from the Tic.

Optional connections

The Tic's **5V (out)** pin provides access to the output of the Tic's 5V regulator, which also powers the Tic's microcontroller and the red and yellow LEDs. You can use the Tic's regulator to power your microcontroller or other serial device if the device does not draw too much current (see **Section 5.8**).

The **VM** pin provides access to the Tic's power supply after the reverse-voltage protection circuit, and this pin can be used to provide reverse-voltage-protected power to other components in the system if the Tic supply voltage is within the operating range of those components. **Note:** this pin should not be used to supply more than 500 mA; higher-current connections should be made directly to the power

supply. Unlike the **5V (out)** pin described above, this is not a regulated, logic-level output.

The **ERR** pin of the Tic is normally pulled low, but drives high to indicate when an error is stopping the motor. You can connect this line to an input on your microcontroller (assuming it is 5V tolerant) to quickly tell whether the Tic is experiencing an error or not. Alternatively, you can query the Tic’s serial interface to see if an error is happening, and which specific errors are happening. For more information about the ERR pin, see **Section 5.4**.

The **$\overline{\text{RST}}$** pin of the Tic is connected directly to the reset pin of the Tic’s microcontroller and also has a 10 kΩ resistor pulling it up to 5 V. You can drive this pin low to perform a hard reset of the Tic’s microcontroller and de-energize the stepper motor, but this should generally not be necessary for typical applications. You should wait at least 10 ms after a reset to transmit to the Tic.

Connecting a serial device to multiple Tics

The Tic’s serial protocol is designed so that you can control multiple Tics using a single TTL serial port. Before attempting to do this, however, we recommend that you first get your system working with just one Tic as described above.

Next, make sure that the serial device and the Tics all share a common ground, for example by connecting a GND pin from the device to a GND pin on each of the Tics. Make sure that the TX pin on the serial device is connected to the RX pin of each Tic (via a level shifter if needed).

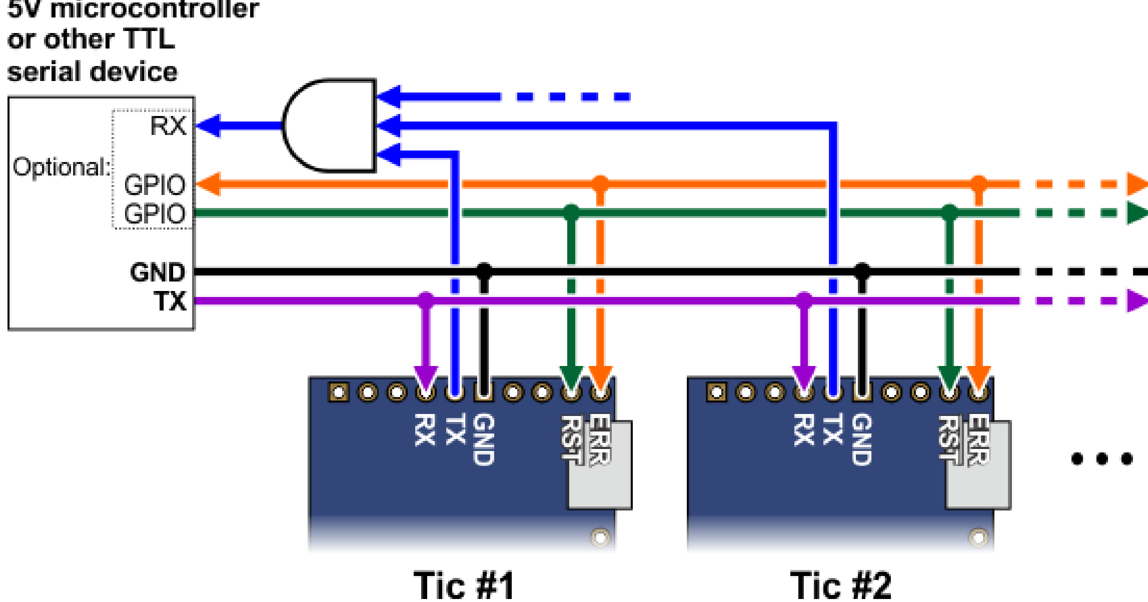
If you attempt to run the SerialSpeedControl example in this configuration, you should see each of your Tic controllers moving their stepper motors in unison. That example uses the Tic’s Compact Protocol, which is only suitable for controlling one device. The Compact Protocol commands do not contain a device number, so every Tic device that sees a Compact Protocol command will obey it. This is probably not what you want.

To allow independent control of multiple Tics, you should use the Tic Control Center to configure each Tic to have a different device number. Then you should change your code to use the Pololu Protocol as described in **Section 9**. If you are using our Tic Arduino library, you can declare one object for each Tic, and specify the device number of each Tic, by writing code like this at the top of your sketch, which uses device numbers 14 and 15:

```
TicSerial tic1(ticSerial, 14);
TicSerial tic2(ticSerial, 15);
```

If you want to read data from multiple Tics, you cannot simply connect all of the Tic TX lines together, because when one of Tics tries to drive its TX line low to send data, the TX lines from the other Tics will still be driving the line high and will prevent the signal from going all the way to 0 V. Instead, you will need to connect an external AND gate. The TX line of each Tic should be connected to an input line of the AND gate. The output of the AND gate should be connected to the RX line of your serial

device (through a voltage divider or level shifter if necessary). The following diagram shows these connections along with optional connections of the ERR and **RST** pins:



The ERR pins can all be safely connected together. In such a configuration, the line will be high if one or more Tics has an error; otherwise, it will be low. Additionally, the Tics are configured by default to treat a high signal on their ERR lines as an error, so an error on one Tic will trigger an error on all other Tics when their ERR lines are connected as shown in the above diagram. This behavior can be disabled by checking the “Ignore ERR line high” box under the “Advanced settings” tab of the Tic Control Center. For more information on the ERR pin and error handling in general, see **Section 5.4**.

Using I²C instead of serial to read data from multiple Tics does not require an AND gate (see **Section 4.6**).

More information about the serial interface

This user’s guide has more information about the Tic’s commands (**Section 8**) and serial protocol (**Section 9**).

4.6. Setting up I²C control

This section explains how to connect a microcontroller to the Tic’s I²C interface so that you can send commands to control the Tic. The **Tic Stepper Motor Controller library for Arduino [https://github.com/pololu/tic-arduino]** makes it particularly easy to control the Tic from an Arduino or Arduino-compatible board such as an **A-Star 32U4 [https://www.pololu.com/a-star]**.

© 2001–2019 Pololu Corporation

About the I²C interface

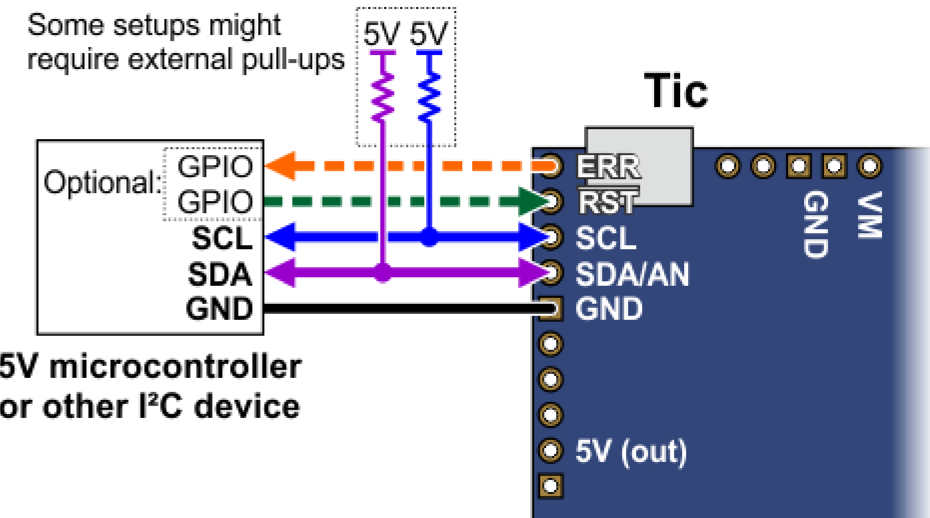
The **SCL** and **SDA/AN** pins of the Tic provide its I²C interface. The pins are open drain outputs, meaning that they only drive low and they never drive high. Each pin has a 220 Ω or 470 Ω series resistor protecting it from short circuits. By default, each pin is pulled up to 5 V by the Tic's microcontroller with a pull-up resistor that is typically around 40 kΩ. When the Tic is reading the SCL or SDA pin as an input, any value over 2.1 V will be considered to be high.

Devices on the I²C bus have two roles: a *master* that initiates communication, and a *slave* that responds to requests from a master. The Tic acts as the slave. The Tic uses a feature of I²C called *clock stretching*, meaning that it holds the SCL line low to delay I²C communication while it is busy processing data from the master.

Connecting an I²C device to one Tic

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor. You should leave your Tic's control mode set to its default value of "Serial / I²C / USB" and leave the "Device number" set to its default value of 14. The "Device number" specifies the 7-bit address for the Tic to use on the I²C bus.

Next, connect your device's SCL pin to the Tic's SCL pin, and connect your device's SDA pin to the Tic's SDA pin. You should also connect your device's GND (ground) pin to one of the Tic's GND pins. These connections, and some other optional connections, are shown in the diagram below:



Because the Tic considers an input value of 2.1 V on SCL or SDA to be high, you can connect those pins directly to 3.3 V microcontrollers without needing a level shifter. If your microcontroller's I²C interface is not 5V tolerant, it will usually still have a diode going from each I/O pin to its logic supply. These diodes clamp the voltage on the pins, preventing the Tic's pull-up resistors from pulling

the pins too high. If you want to be extra safe and not rely on the clamping diodes, you can disable the Tic’s pull-up resistors by going to the “Advanced settings” tab in the Tic Control Center, changing the functions of SCL and SDA to “Serial”, and making sure the “Pull-up” checkbox for each pin is not checked.

Depending on your setup, you might need to add pull-up resistors to the SCL and SDA lines of your I²C bus to ensure that the signals rise fast enough. The Tic’s pull-up resistors are enabled by default, and many I²C master devices will have pull-ups too, but that might not be enough, especially if you want to use speeds faster than 100 kHz or have long wires. The **I²C-bus specification and user manual [https://www.pololu.com/file/0J435/UM10204.pdf]** (1MB pdf) has some information about picking pull-up resistors in the “Pull-up resistor sizing” section, and trying a value around 10 kΩ is generally a good starting point.

If you are using an Arduino or Arduino-compatible board, you should now try running the I2CSpeedControl example that comes with the **Tic Arduino library [https://github.com/pololu/tic-arduino]**. If you are using a different kind of microcontroller board, you will need to find or write code to control the Tic on your platform. If you are writing your own code, we recommend that you first learn how to use the I²C interface of your platform, and then use the I2CSpeedControl example as a reference. You should also refer to the sections in this guide about the Tic’s commands (**Section 8**) and I²C protocol (**Section 10**).

If your connections and code are OK, you should now see the Tic’s stepper motor moving back and forth. If the stepper motor is not moving, you should check all of your connections and solder joints (if applicable). You should check the “Status” tab of the Tic Control Center to see if any errors are being reported. You can also try slowing down your I²C clock speed to something very slow like 1 kHz or 10 kHz. If slowing down the clock works, then the problem might be due to not having strong enough pull-up resistors on the SDA and SCL lines.

Optional connections

The Tic’s **5V (out)** pin provides access to the output of the Tic’s 5V regulator, which also powers the Tic’s microcontroller and the red and yellow LEDs. You can use the Tic’s regulator to power your microcontroller or another device if the device does not draw too much current (see **Section 5.8**).

The **VM** pin provides access to the Tic’s power supply after the reverse-voltage protection circuit, and this pin can be used to provide reverse-voltage-protected power to other components in the system if the Tic supply voltage is within the operating range of those components. **Note:** this pin should not be used to supply more than 500 mA; higher-current connections should be made directly to the power supply. Unlike the **5V (out)** pin described above, this is not a regulated, logic-level output.

The **ERR** pin of the Tic is normally pulled low, but drives high to indicate when an error is stopping the motor. You can connect this line to an input on your microcontroller (assuming it is 5V tolerant) to

quickly tell whether the Tic is experiencing an error or not. Alternatively, you can query the Tic’s serial interface to see if an error is happening, and which specific errors are happening. For more information about the ERR pin, see **Section 5.4**.

The $\overline{\textbf{RST}}$ pin of the Tic is connected directly to the reset pin of the Tic’s microcontroller and also has a 10 kΩ resistor pulling it up to 5 V. You can drive this pin low to perform a hard reset of the Tic’s microcontroller and de-energize the stepper motor, but this should generally not be necessary for typical applications. You should wait at least 10 ms after a reset to transmit to the Tic.

Controlling multiple Tics with I^{2}C

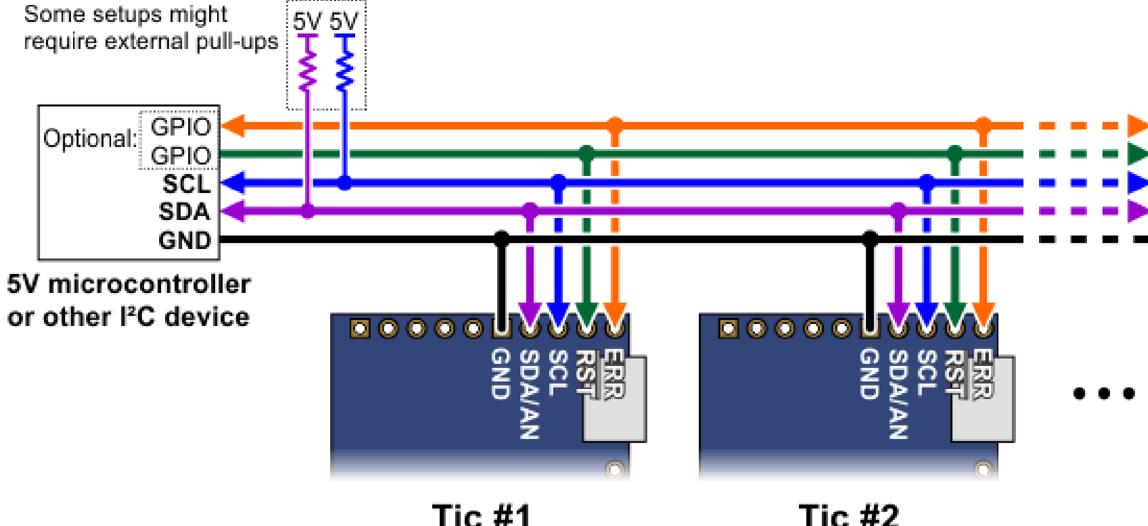
I^{2}C is designed so that you can control multiple slave devices on a single bus. Before attempting to do this, however, we recommend that you first get your system working with just one Tic as described above.

Next, make sure that the I^{2}C master device and the Tics all share a common ground, for example by connecting a GND pin from the I^{2}C master device to a GND pin on each of the Tics. Make sure that the SCL pins of all devices are connected and that the SDA pins of all devices are connected.

You should use the Tic Control Center to configure each Tic to have a different “Device number”, which specifies the 7-bit I^{2}C address to use. Then you should change your code to use those addresses. If you are using our Tic Arduino library, you can declare one object for each Tic by writing code like this at the top of your sketch, which uses addresses 14 and 15:

```
TicI2C tic1(14);
TicI2C tic2(15);
```

The following diagram shows the standard I^{2}C connections described above along with optional connections of the ERR and $\overline{\textbf{RST}}$ pins:



The ERR pins can all be safely connected together. In such a configuration, the line will be high if one or more Tics has an error; otherwise, it will be low. Additionally, the Tics are configured by default to treat a high signal on their ERR lines as an error, so an error on one Tic will trigger an error on all other Tics when their ERR lines are connected as shown in the above diagram. This behavior can be disabled by checking the “Ignore ERR line high” box under the “Advanced settings” tab of the Tic Control Center. For more information on the ERR pin and error handling in general, see **Section 5.4**.

More information about the I²C interface

This user’s guide has more information about the Tic’s commands (**Section 8**) and I²C protocol (**Section 10**).

4.7. Setting up RC position control

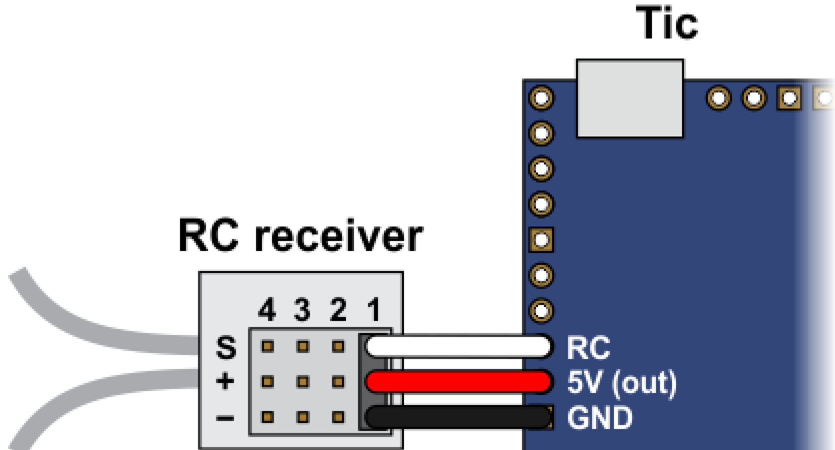
This section explains how to set up the Tic to read a hobby RC servo signal and use that signal to control the position of the stepper motor.

In this mode, the Tic will behave somewhat like an RC servo. One advantage of the Tic is that you can set up the range of motion of your system to be very large (e.g. tens of turns) or very small (e.g. a few degrees). One important difference between the Tic and an RC servo is that the Tic does not receive any kind of feedback from the stepper motor about its position. When you power on an RC servo and send it a signal, it immediately moves to the position corresponding to that signal. When you power on the Tic, it does not know what position the stepper motor is in, so it will wait for a valid RC signal, and then assume that the stepper motor is already at the position that corresponds to that signal. Also, there are other error conditions besides losing power that will cause the Tic to become uncertain about its current position and re-learn it when the system returns to normal (see **Section 5.4**).

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test

© 2001–2019 Pololu Corporation

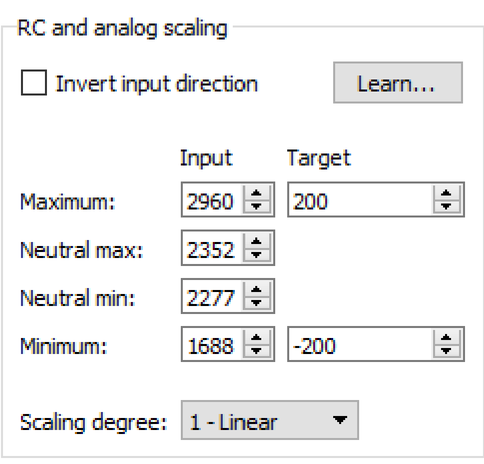
your stepper motor. Next, with the system unpowered, connect your RC receiver to the Tic's GND, 5V, and RC pins as shown in the diagram below.



In this configuration, the RC receiver will be powered by the Tic's 5 V regulator via the 5V output pin. If you want to power the receiver from another power source instead, you should **not** connect the Tic's 5V pin to it as doing so would short the two sources together and could damage the Tic or receiver.

If the Tic gets reset when you plug in your RC receiver, it might be because the in-rush current of the receiver is too much for the Tic's 5V line and causes its voltage to drop temporarily. As general good engineering practice, we recommend making and breaking electrical connections only while your devices are powered off.

© 2001–2019 Pololu Corporation



Example Tic scaling settings for RC position control mode.

Now connect the Tic to your computer via USB. In the Tic Control Center software, set the Tic's control mode to "RC position" and click "Apply settings". In the "Scaling" box, click "Learn…" to start the Input Setup Wizard. The wizard will help you measure the neutral, maximum, and minimum positions of your RC signal. When the wizard is finished, it will set five of the input scaling parameters (input maximum, input neutral max, input neutral min, input minimum, and invert input direction) appropriately so that the neutral RC signal gets mapped to a position of 0, the maximum RC signal gets mapped to the target maximum, and the minimum RC signal gets mapped to the target minimum. If you have previously changed the target maximum and target minimum, you should set them back to their default values of 200 and -200, respectively. Click "Apply settings" to save these settings to the Tic.

Now connect motor power and click "Resume" to start your system. If you move your input from the neutral position to the maximum position, you should see the motor move by 200 steps. If you move your input from the neutral position to the minimum position, you should see the motor move by 200 steps in the other direction.

You should make sure that the motor is moving in the correct direction. If it is not, you can check the "Invert motor direction" checkbox to fix it. (You could also rewire the stepper motor to reverse the current in one coil, but be sure to turn off the stepper motor power before doing that.)

Next, you should set the target maximum and minimum parameters in the "Scaling" box to set the range of motion of your system. The target maximum must be zero or more, and the target minimum must zero or less. These numbers correspond to microsteps if you have enabled microstepping.

Finally, check the "Scaling degree" parameter. The default setting is "1 – Linear". If you want finer control near the neutral point of your input and coarser control near the ends, you can change it to one of the higher settings.

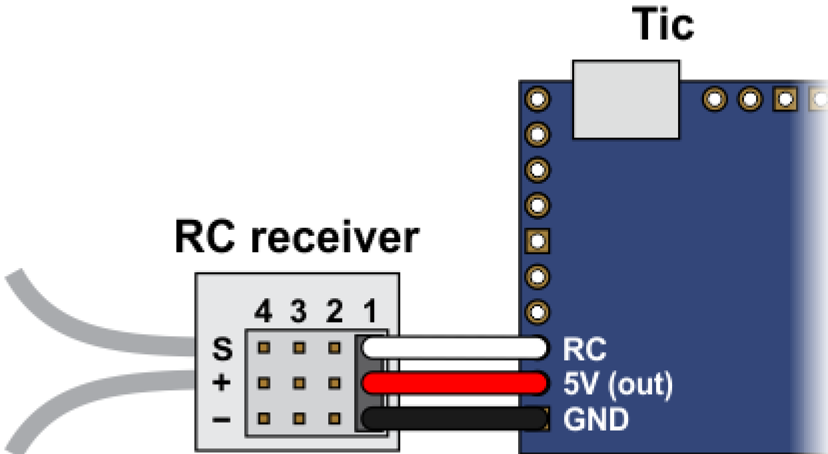
For details about what kind of RC signals the Tic can accept, and how the input scaling works, see **Section 5.2**.

4.8. Setting up RC speed control

This section explains how to set up the Tic to read a hobby RC servo signal and use that signal to control the speed of the stepper motor.

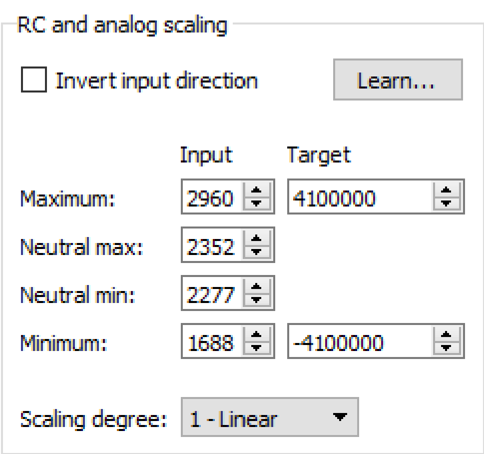
In this mode, the Tic will behave like an electronic speed controller (ESC), except with a stepper motor instead of a DC motor.

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor. Next, with the system unpowered, connect your RC receiver to the Tic's GND, 5V, and RC pins as shown in the diagram below.



In this configuration, the RC receiver will be powered by the Tic's 5 V regulator via the 5V output pin. If you want to power the receiver from another power source instead, you should **not** connect the Tic's 5V pin to it as doing so would short the two sources together and could damage the Tic or receiver.

If the Tic gets reset when you plug in your RC receiver, it might be because the in-rush current of the receiver is too much for the Tic's 5V line and causes its voltage to drop temporarily. As general good engineering practice, we recommend making and breaking electrical connections only while your devices are powered off.



Example Tic scaling settings for RC speed control mode.

Now connect the Tic to your computer via USB. In the Tic Control Center software, set the Tic's control mode to "RC speed" and click "Apply settings". In the "Scaling" box, click "Learn…" to start the Input Setup Wizard. The wizard will help you measure the neutral, maximum, and minimum positions of your RC signal. When the wizard is finished, it will set five of the input scaling parameters (input maximum, input neutral max, input neutral min, input minimum, and invert input direction) appropriately so that the neutral RC signal gets mapped to a velocity of 0, the maximum RC signal gets mapped to the target maximum, and the minimum RC signal gets mapped to the target minimum. You should change the target maximum to be equal to the maximum velocity that you want your motor to move in the forward/positive direction. Since you already set the "Max speed" parameter in the "Motor" box (see **Section 4.3**), you could just copy that value into the target maximum box. If you want your motor to go the same speed in both directions, you should set the target minimum to the negative of the target maximum. Otherwise, you should set the target minimum to be the lowest (most negative) velocity that you want your motor to have when moving in the other direction. Click "Apply settings" to save these settings to the Tic.

Now connect motor power and click "Resume" to start your system. If the Tic Control Center software says "Motor de-energized because of safe start violation.", you should center your input. After doing that, you should be able to move your RC input to control the speed of the motor.

The safe-start feature helps prevent unexpected motion of the stepper motor by making sure that the stepper motor does not start moving until after your RC input goes to the neutral position. When you are starting up your system, you will always have to move your input to the neutral position if it was not there already. It is possible to disable this feature by checking the "Disable safe start" checkbox in the "Advanced settings" tab.

You should make sure that the motor is moving in the correct direction. If it is not, you can check the "Invert motor direction" checkbox to fix it. (You could also rewire the stepper motor to reverse the current in one coil, but be sure to turn off the stepper motor power before doing that.)

Finally, check the "Scaling degree" parameter. The default setting is "1 – Linear". If you want finer control at low speeds and coarser control at high speeds, you can change it to one of the higher settings.

For details about what kind of RC signals the Tic can accept, and how the input scaling works, see

Section 5.2.

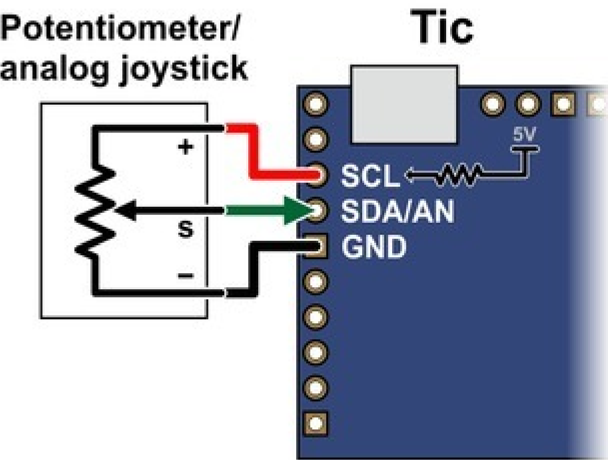
4.9. Setting up analog position control

This section explains how to set up the Tic to read an analog input and use that signal to control the position of the stepper motor.

It is important to note that the Tic does not receive any kind of feedback from the stepper motor about its position. When you power on the Tic, it does not know what position the stepper motor is in, so it will read the analog input and then assume that the stepper motor is already at the position that corresponds to that input. Also, there are other error conditions besides losing power that will cause the Tic to become uncertain about its current position from the analog input when the system returns to normal (see **Section 5.4**).

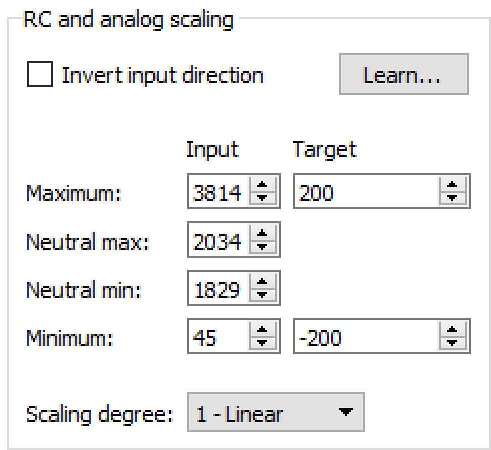
If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor. Next, with the system unpowered, connect your analog signal to the Tic as described below.

If you are using a potentiometer to make the analog signal, you should connect the potentiometer's wiper to SDA/AN and connect the other two ends to GND and SCL, as shown in the diagram below. In analog mode, the SCL line is driven high (5 V) to power the potentiometer (note that the SCL pin is protected by a 220 Ω or 470 Ω series resistor, so it will not be damaged by inadvertent shorts to ground).



If you are using something other than a potentiometer to generate the analog signal, make sure that the ground node of that device is connected to a GND pin on the Tic, and that the analog signal from that device is connected to the Tic's SDA/AN line. The Tic's analog input can only accept signals between 0 V and 5 V with respect to GND; signals outside of this range could damage the Tic.

Now connect the Tic to your computer via USB. In the Tic Control Center software, set the Tic's control mode to "Analog position" and click "Apply settings". In the "Scaling" box, click "Learn…" to start the Input Setup Wizard. The wizard will help you measure the neutral, maximum, and minimum positions of your analog signal. When the wizard is finished, it will set five of the input scaling parameters (input maximum, input neutral max, input neutral min, input minimum, and invert input direction) appropriately so that the neutral analog signal gets mapped to a position of 0, the maximum analog signal gets mapped to the target maximum, and the minimum analog signal gets mapped to the target minimum. If you have previously changed the target maximum and target minimum, you should set them back to their default values of 200 and -200, respectively. Click "Apply settings" to save these settings to the Tic.



Example Tic scaling settings for analog position control mode.

Now connect motor power and click "Resume" to start your system. If you move your input from the neutral position to the maximum position, you should see the motor move by 200 steps. If you move your input from the neutral position to the minimum position, you should see the motor move by 200 steps in the other direction.

You should make sure that the motor is moving in the correct direction. If it is not, you can check the "Invert motor direction" checkbox to fix it. (You could also rewire the stepper motor to reverse the current in one coil, but be sure to turn off the stepper motor power before doing that.)

Next, you should set the target maximum and minimum parameters in the "Scaling" box to set the range of motion of your system. The target maximum must be zero or more, and the target minimum must zero or less. These numbers correspond to microsteps if you have enabled microstepping.

Finally, check the "Scaling degree" parameter. The default setting is "1 – Linear". If you want finer control near the neutral point of your input and coarser control near the ends, you can change it to one

of the higher settings.

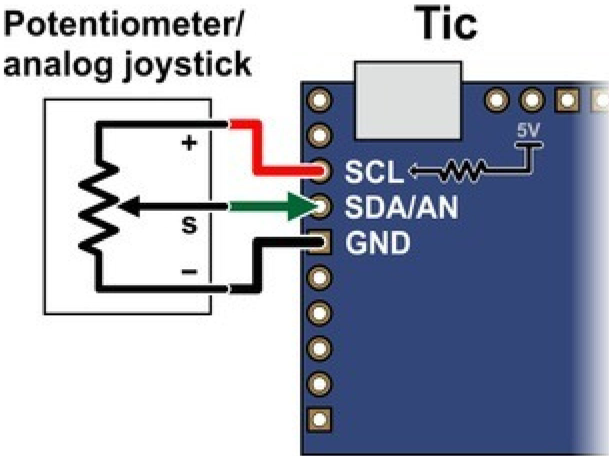
For details about how the input scaling works, see **Section 5.2**.

4.10. Setting up analog speed control

This section explains how to set up the Tic to read an analog input and use that signal to control the speed of the stepper motor.

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor. Next, with the system unpowered, connect your analog signal to the Tic as described below.

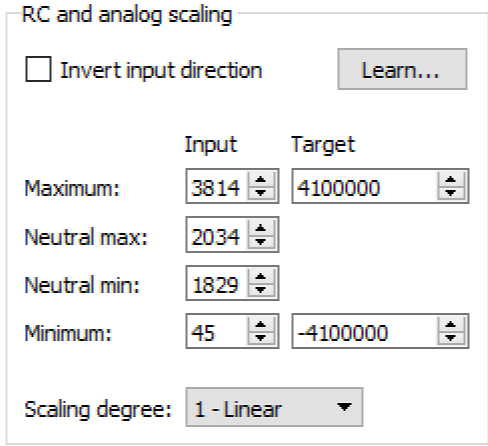
If you are using a potentiometer to make the analog signal, you should connect the potentiometer’s wiper to SDA/AN and connect the other two ends to GND and SCL, as shown in the diagram below. In analog mode, the SCL line is driven high (5 V) to power the potentiometer (note that the SCL pin is protected by a 220 Ω or 470 Ω series resistor, so it will not be damaged by inadvertent shorts to ground).



If you are using something other than a potentiometer to generate the analog signal, make sure that the ground node of that device is connected to a GND pin on the Tic, and that the analog signal from that device is connected to the Tic’s SDA/AN line. The Tic’s analog input can only accept signals between 0 V and 5 V with respect to GND; signals outside of this range could damage the Tic.

Now connect the Tic to your computer via USB. In the Tic Control Center software, set the Tic’s control mode to “Analog speed” and click “Apply settings”. In the “Scaling” box, click “Learn…” to start the Input Setup Wizard. The wizard will help you measure the neutral, maximum, and minimum positions of your analog signal. When the wizard is finished, it will set five of the input scaling parameters (input maximum, input neutral max, input neutral min, input minimum, and invert input direction)

appropriately so that the neutral analog signal gets mapped to a velocity of 0, the maximum analog signal gets mapped to the target maximum, and the minimum analog signal gets mapped to the target minimum. You should change the target maximum to be equal to the maximum velocity that you want your motor to move in the forward/positive direction. Since you already set the "Max speed" parameter in the "Motor" box (see **Section 4.3**), you could just copy that value into the target maximum box. If you want your motor to go the same speed in both directions, you should set the target minimum to the negative of the target maximum. Otherwise, you should set the target minimum to be the lowest (most negative) velocity that you want your motor to have when moving in the other direction. Click "Apply settings" to save these settings to the Tic.



Example Tic scaling settings for analog speed control mode.

Now connect motor power and click "Resume" to start your system. If the Tic Control Center software says "Motor de-energized because of safe start violation.", you should center your input. After doing that, you should be able to move your analog input to control the speed of the motor.

The safe-start feature helps prevent unexpected motion of the stepper motor by making sure that the stepper motor does not start moving until after your analog input goes to the neutral position. When you are starting up your system, you will always have to move your input to the neutral position if it was not there already. It is possible to disable this feature by checking the "Disable safe start" checkbox in the "Advanced settings" tab.

You should make sure that the motor is moving in the correct direction. If it is not, you can check the "Invert motor direction" checkbox to fix it. (You could also rewire the stepper motor to reverse the current in one coil, but be sure to turn off the stepper motor power before doing that.)

Finally, check the "Scaling degree" parameter. The default setting is "1 – Linear". If you want finer control at low speeds and coarser control at high speeds, you can change it to one of the higher

settings.

For details about how the input scaling works, see **Section 5.2**.

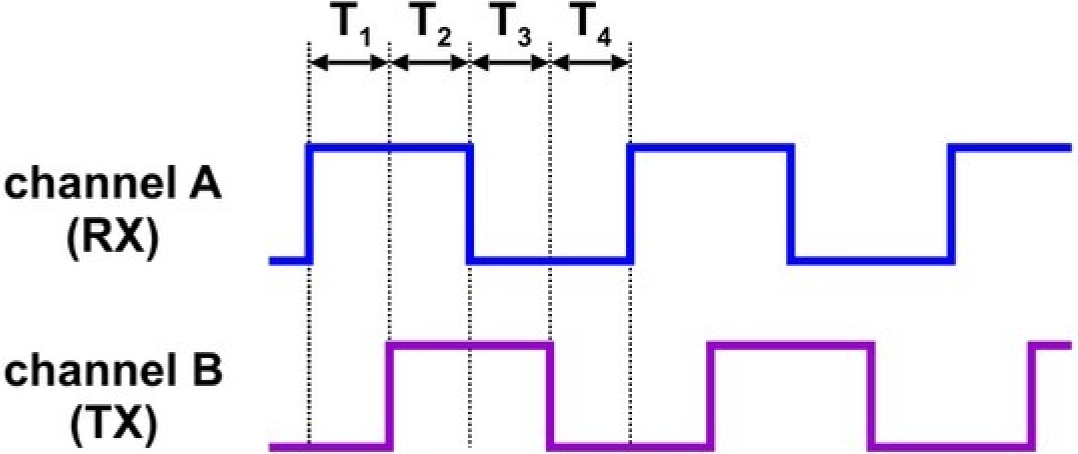
4.11. Setting up encoder position control

This section explains how to set up the Tic to read signals from a quadrature encoder and use them to control the position of the stepper motor.

The Tic does **not** support closed-loop control with encoder feedback. The Tic's encoder input is meant to be connected to a rotary encoder that is turned by hand. If you have a stepper motor with an integrated encoder, you should not try to connect the motor's encoder to the Tic.

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor.

When in encoder mode, the Tic uses its RX and TX lines as encoder inputs. Each of these lines has an integrated 100 kΩ resistor pulling it up to 5 V and a 220 Ω or 470 Ω series resistor protecting it from short circuits (e.g. in case it is inadvertently put into serial mode, which uses TX as an output, with an encoder still connected). The Tic expects to see standard quadrature encoder signals like this on its encoder inputs:

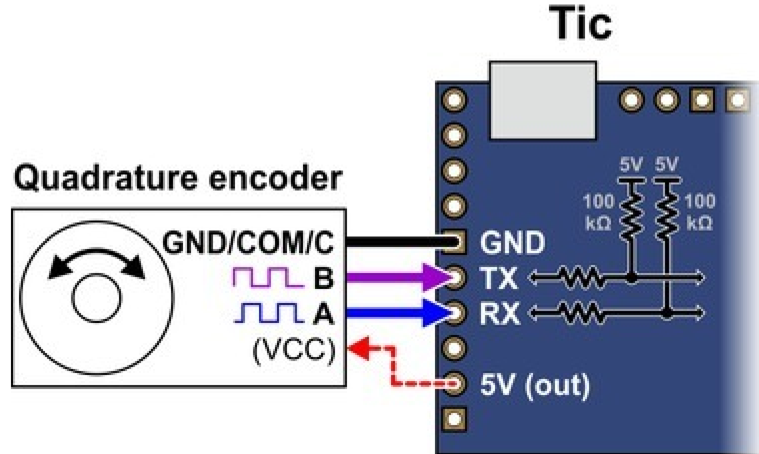


The time between the channel transitions (labeled $T_{1\text{-}4}$ in the diagram above) must be at least 100 μs.

For this control interface, we generally recommend common three-pin mechanical rotary encoders with quadrature outputs, though other kinds of quadrature encoders can also be used. A three-pin

rotary encoder has two signal pins, A and B, which should be connected to RX and TX on the Tic, and a common pin (sometimes labeled "C") that should be connected to ground. Note that the common pin is often in the middle, but you should always refer to your encoder documentation to identify which pin is which. These encoders do not require power as the signal pins just alternate between floating and ground as the dial is rotated. The built-in pull-ups on the RX and TX pins make the signal high during the times when the encoder outputs are floating, so there is no need for external pull-ups. Other kinds of quadrature encoders might require power, and the 5V output on the Tic can be used to power them if their documentation indicates they can operate at 5 V.

As a first step, you should turn off the power to your system and then connect your encoder to the Tic as described above and shown in the diagram below. You can swap your A and B connections to flip the direction of the encoder.



Now connect the Tic to your computer via USB. In the Tic Control Center software, set the Tic's control mode to "Encoder position". If you have previously changed the target maximum and target minimum, you should set them back to their default values of 200 and −200, respectively. If you have previously changed the encoder prescaler or postscaler settings, you should change them both back to their default value of 1. Also, make sure that the "Enable unbounded position control" setting for the encoders is unchecked, as it is by default. Click "Apply settings" to save these settings to the Tic.

Now connect motor power to start your system running. As you turn your encoder, you should see your stepper motor moving proportionally: one count from the encoder corresponds to one step from the Tic.

If your motor is not moving as you turn the encoder, you should look at the message at the bottom of the Tic Control Center and also check for errors in Status tab. If there are any errors, you should address them before continuing. If the system is still not working, you should look at the "Encoder position" displayed in the Status tab, which is the raw count from your encoder. This number is pinned

to zero when motor power is off, so make sure you have connected your motor power to the Tic. If you turn your encoder one way, this number should go up. If you turn it the other way, this number should go down. If the “Encoder position” is not changing, or only changing by one count, it is possible that your encoder is not wired correctly. Check all of your connections and soldering joints (if applicable). If you have access to an oscilloscope, you should check the signals on RX and TX. If the “Encoder position” is responding properly to the encoder but the “Input after scaling” variable is not, then make sure you have set those settings to their defaults as described above.

The Tic expects that transitions on its encoder inputs will be at least 100 μs apart. If your encoder signal is faster than this, the Tic might miss some encoder counts or could even measure counts in the wrong direction. To see whether this is happening, you should try turning your encoder as fast as you expect it to be turned in your application. As you do this, look at the “Encoder skip” count shown in the “Errors” box in the “Status” tab. If the count goes up when you turn the encoder, that means the Tic is missing some encoder counts. In that case, if encoder accuracy is important in your application, you might consider getting a different encoder or turning your encoder more slowly.

You should make sure that the motor is moving in the correct direction. If it is not, you can swap the RX and TX connections or check the “Invert motor direction” checkbox to fix it. (You could also rewire the stepper motor to reverse the current in one coil, but be sure to turn off the stepper motor power before doing that.)

Next, you should use the encoder prescaler and postscaler to specify how far the stepper motor should move when you turn the encoder. Every time the encoder position changes by the prescaler amount, the “Input after scaling” variable will be changed by the postscaler amount. So if you increase the prescaler from its default value of 1, it will take more encoder movement to get the same amount of movement from the stepper motor. If you increase the postscaler from its default value of 1, the stepper motor will move further for the same amount of movement from the encoder. If your encoder has detents, it usually makes sense to set the encoder prescaler to the number of counts you get per detent, which is typically 4.

Finally, you should set the target maximum and minimum parameters in the “Scaling” box to set the range of motion of your system. The target maximum must be zero or more, and the target minimum must zero or less. These numbers are denominated in microsteps if you have enabled microstepping. Note that these are the only two numbers in the “Scaling” box that have an effect in encoder mode. Alternatively, if you want your system’s range to be unlimited, check the “Enable unbounded position control” checkbox, in which case all the numbers in the “Scaling” box will be ignored.

For details about how the Tic’s encoder input works, see **Section 5.3**.

4.12. Setting up encoder speed control

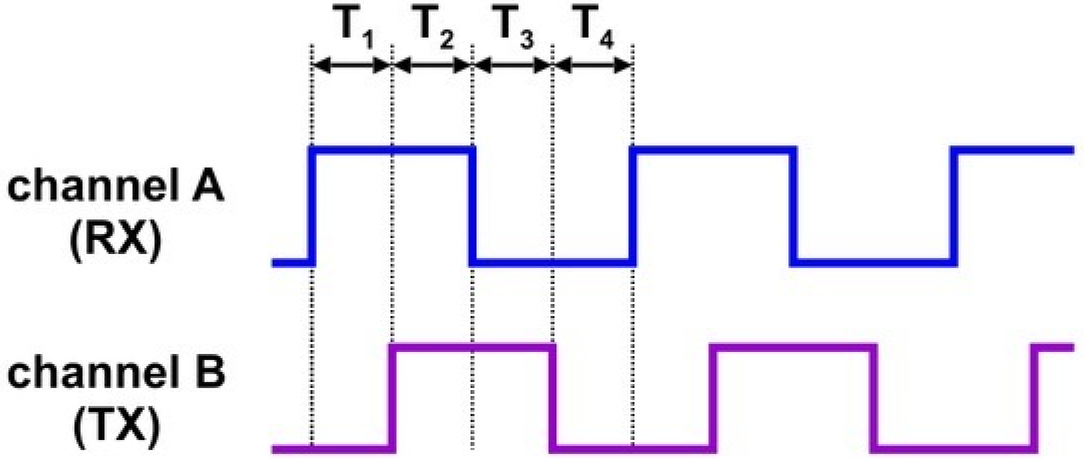
This section explains how to set up the Tic to read a quadrature encoder signal and use that signal to

control the speed of the stepper motor.

The Tic does **not** support closed-loop control with encoder feedback. The Tic's encoder input is meant to be connected to a rotary encoder that is turned by hand. If you have a stepper motor with an integrated encoder, you should not try to connect the motor's encoder to the Tic.

If you have not done so already, you should follow the instructions in **Section 4.3** to configure and test your stepper motor.

When in encoder mode, the Tic uses its RX and TX lines as encoder inputs. Each of these lines has an integrated 100 kΩ resistor pulling it up to 5 V and a 220 Ω or 470 Ω series resistor protecting it from short circuits (e.g. in case it is inadvertently put into serial mode, which uses TX as an output, with an encoder still connected). The Tic expects to see standard quadrature encoder signals like this on its encoder inputs:

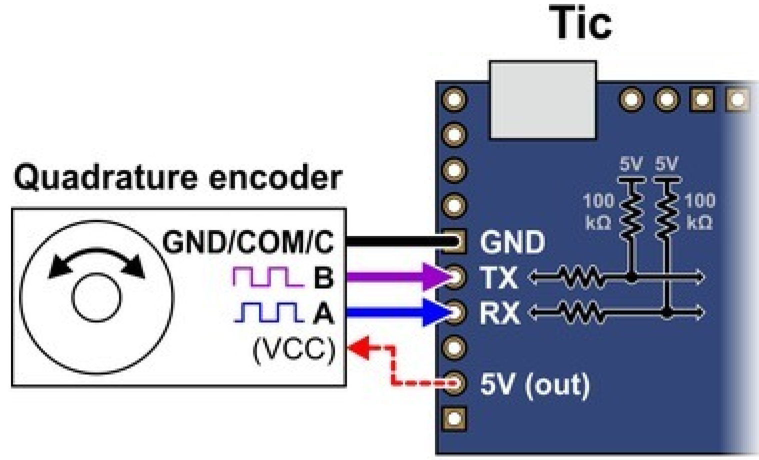


The time between the channel transitions (labeled $T_{1\text{-}4}$ in the diagram above) must be at least 100 μs.

For this control interface, we generally recommend common three-pin mechanical rotary encoders with quadrature outputs, though other kinds of quadrature encoders can also be used. A three-pin rotary encoder has two signal pins, A and B, which should be connected to RX and TX on the Tic, and a common pin (sometimes labeled "C") that should be connected to ground. Note that the common pin is often in the middle, but you should always refer to your encoder documentation to identify which pin is which. These encoders do not require power as the signal pins just alternate between floating and ground as the dial is rotated. The built-in pull-ups on the RX and TX pins make the signal high during the times when the encoder outputs are floating, so there is no need for external pull-ups. Other kinds of quadrature encoders might require power, and the 5V output on the Tic can be used to power them

if their documentation indicates they can operate at 5 V.

As a first step, you should turn off the power to your system and then connect your encoder to the Tic as described above and shown in the diagram below. You can swap your A and B connections to flip the direction of the encoder.



Now connect the Tic to your computer via USB. In the Tic Control Center software, set the Tic's control mode to "Encoder speed". In the "Scaling" box, you should change the target maximum to be equal to the maximum velocity that you want your motor to move in the forward/positive direction. Since you already set the "Max speed" parameter in the "Motor" box (see **Section 4.3**), you could just copy that value into the target maximum box. If you want your motor to go the same speed in both directions, you should set the target minimum to the negative of the target maximum. Otherwise, you should set the target minimum to be the lowest (most negative) velocity that you want your motor to have when moving in the other direction. If you have previously changed the encoder prescaler setting, you should change it back to its default value of 1. You should set the encoder postscaler to approximately one hundredth of the "Max speed". Click "Apply settings" to save these settings to the Tic.

Now connect motor power to start your system running. As you turn your encoder, you should see the speed of your stepper motor changing proportionally: moving the encoder by one count should change the target velocity by the value you entered for the encoder postscaler.

If your motor speed is not changing as you turn the encoder, you should look at the message at the bottom of the Tic Control Center and also check for errors in Status tab. If there are any errors, you should address them before continuing. If the system is still not working, you should look at the "Encoder position" displayed in the Status tab, which is the raw count from your encoder. This number is pinned to zero when motor power is off, so make sure you have connected your motor power to the Tic. If you turn your encoder one way, this number should go up. If you turn it the other way, this number should go down. If the "Encoder position" is not changing, or only changing by one count, it is

possible that your encoder is not wired correctly. Check all of your connections and soldering joints (if applicable). If you have access to an oscilloscope, you should check the signals on RX and TX. If the "Encoder position" is responding properly to the encoder but the "Input after scaling" variable is not, then make sure you set the Tic's input settings as described above. The Tic expresses velocities in units of pulses per 10,000 seconds, so if the "Input after scaling" is a non-zero number that is much smaller than 10,000, your motor might be moving, but it would be moving too slowly to be easily noticed. Make sure you have set the encoder postscaler as described above.

The Tic expects that transitions on its encoder inputs will be at least 100 μs apart. If your encoder signal is faster than this, the Tic might miss some encoder counts or could even measure counts in the wrong direction. To see whether this is happening, you should try turning your encoder as fast as you expect it to be turned in your application. As you do this, look at the "Encoder skip" count shown in the "Errors" box in the "Status" tab. If the count goes up when you turn the encoder, that means the Tic is missing some encoder counts. In that case, if encoder accuracy is important in your application, you might consider getting a different encoder or turning your encoder more slowly.

You should make sure that the motor is moving in the correct direction. If it is not, you can swap the RX and TX connections or check the "Invert motor direction" checkbox to fix it. (You could also rewire the stepper motor to reverse the current in one coil, but be sure to turn off the stepper motor power before doing that.)

Finally, you should set the encoder prescaler and postscaler to specify how much the stepper motor speed should change as you turn the encoder. Every time the encoder position changes by the prescaler amount, the "Input after scaling" variable will be changed by the postscaler amount. So if you increase the prescaler from its default value of 1, it will take more encoder movement to get the same change in speed from the stepper motor. If your encoder has detents, it usually makes sense to set the encoder prescaler to the number of counts you get per detent, which is typically 4. If you decrease the postscaler, you will have finer control of the motor speed and it will take more turns of the encoder to reach full speed. If you increase the postscaler, you will have coarser control over the motor speed, and it will take fewer turns to reach full speed.

For details about how the Tic's encoder input works, see **Section 5.3**.

4.13. Setting up STEP/DIR control

You can set the Tic's control mode to "STEP/DIR" to turn the Tic into a digitally-configurable **stepper motor driver** **[https://www.pololu.com/category/120/stepper-motor-drivers]**.

In this mode, you will need to connect a microcontroller to the Tic's STEP and DIR pins to control the stepper motor. The STEP and DIR pins are connected through 220 Ω or 470 Ω protection resistors to the STEP and DIR inputs on the Tic's on-board stepper motor driver IC; using them to control the driver bypasses the Tic's speed-limiting and acceleration-limiting features, and the Tic will have no

knowledge of the current position or speed of the stepper motor.

The STEP and DIR lines are pulled down by default. The driver takes one step whenever it sees a rising edge on the STEP pin, and the direction of the step is specified by the DIR pin. For detailed specifications of the STEP/DIR interface, see the **MP6500 datasheet** **[https://www.pololu.com/file/0J1447/MP6500_r1.0.pdf]** (1MB pdf) for the Tic T500, the **DRV8834 datasheet** **[https://www.pololu.com/file/0J617/drv8834.pdf]** (2MB pdf) for the Tic T834, the **DRV8825 datasheet** **[https://www.pololu.com/file/0J590/drv8825.pdf]** (1MB pdf) for the Tic T825, or the **TB67S249FTG datasheet** **[https://www.pololu.com/file/0J1523/TB67S249FTG_datasheet_en_20170818.pdf]** (533k pdf) for the Tic T249.

In STEP/DIR mode, the Tic's USB, serial, and I²C interfaces can still be used to set the driver's current limit, decay mode, and step mode, or to de-energize the driver.

Alternatively, when the Tic's control mode is set to anything other than "STEP/DIR", you can use the STEP and DIR pins as outputs to control an external stepper motor driver. However, you will still need to supply power to the Tic's VIN pin, or else the Tic will report a "Low VIN" error and not attempt to drive the motor.

4.14. Setting up limit switches and homing

This section explains how to set up limit switches and homing. Limit switches can help prevent your system from leaving its desired range of motion. The homing feature enables the Tic to use a limit switch as a reference to determine the position of the motor. The homing feature requires at least one limit switch, but you can use limit switches without using homing.

To use limit switches, you should first make sure that you have version **1.7.0 or later** of the Tic configuration software. You can see the software version number in the Tic Control Center by opening the "Help" menu and selecting "About". (On macOS, "About" is in the "Pololu Tic Control Center" menu instead.) You can get the latest software from **Section 3**. You will also need to make sure your Tic has firmware version **1.06 or later**. The firmware version is displayed in the "Status" tab of the Tic Control Center. If you have an earlier version of the firmware, see **Section 5.7** for upgrade instructions.

Any of the Tic control pins (SCL, SDA, TX, RX, or RC) can be configured as a digital input for a limit switch.

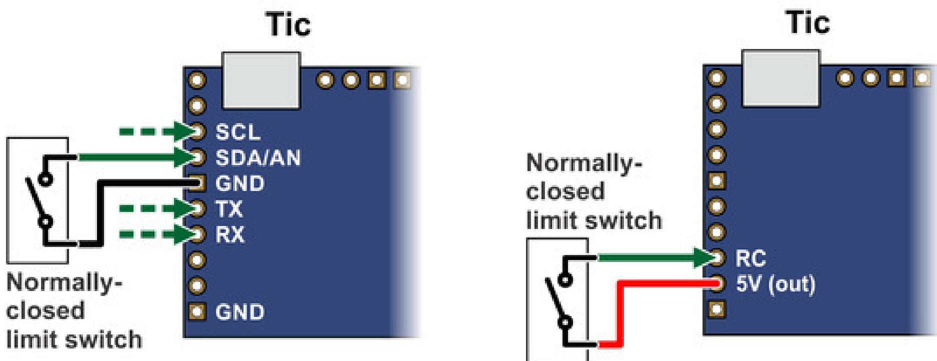
To set up a limit switch, you should go the "Advanced settings" tab and set the function of a pin to "Limit switch forward" or "Limit switch reverse". A forward limit switch prevents the motor from going in the forward direction (increasing position, positive velocity), and a reverse limit switch prevents the motor from going in the reverse direction. The "Active high" checkbox controls the polarity of the limit switch input. If "Active high" is checked, then motor movement will be limited when the voltage on the

pin is high (5 V). If it is not checked, then the motor movement will be limited when voltage on the pin is low (0 V). The “Analog” pin configuration checkbox does not affect the limit switch or homing functionality. It is OK to configure multiple pins to be the same type of limit switch.

We generally recommend using a fail-safe configuration; if the limit switch somehow becomes disconnected from the Tic, then the Tic should treat the limit switch as if it were active. To achieve a fail-safe limit switch configuration on the Tic’s SCL or SDA pins, check the pin’s “Pull-up” checkbox and its “Active high” checkbox, so that the Tic will pull the line high and consider high to be the active state of the limit switch. To achieve a fail-safe configuration on the TX or RX pins, both of which are always pulled up, check the pin’s “Active high” checkbox. To achieve a fail-safe configuration on the Tic’s RC pin, leave the “Active high” checkbox unchecked.

Once you have configured your limit switch, click the “Apply settings” button. If you are using a fail-safe configuration and have not yet connected the limit switch, the Tic should report that the limit switch is active. You should check that this is the case by looking in the “Status” tab of the Tic Control Center. Find the “Limit switches active” field and make sure it indicates that your limit switch is active. Depending on whether the forward or reverse limit switches are active, this field will either say “Forward”, “Reverse”, or “Both”.

Next, disconnect the Tic from all power sources, including USB, and connect your limit switches. If you are using a fail-safe configuration, you will need to use two contacts of the switch that are normally closed (NC), meaning that electricity conducts between the two contacts when the switch is not activated. You should connect one of the switch contacts to the Tic’s configured limit switch pin, and connect the other switch contact to either GND or 5V, whichever is the opposite of the pin’s default state. This is shown in the diagrams below:



Once you have configured and connected the limit switches, you can test them. Connect the Tic to USB and activate the limit switches (e.g. press them with your hands) while looking at the “Limit switches active” field in the “Status” tab. By default, it should say “None”. When you activate a limit switch, the field should indicate which limit switch direction has been activated.

Next, you can reconnect the stepper motor power and try controlling the motor. Make sure that reverse movement (negative speed, decreasing position) causes movement towards the reverse limit switch if you have one. Make sure that forward movement (positive speed, increasing position) causes movement towards the forward limit switch if you have one.

If appropriate for your system, we recommend further testing: make sure that the motor stops properly when it hits a limit switch, and make sure you can continue operation by driving it away from the limit switch. If the control mode you are using does not allow the stepper motor to hit the limit switches, you might consider temporarily changing the control mode to "Serial / I²C / USB" so you can control the motor manually using the "Set target" interface at the bottom of the "Status" tab.

Setting up homing

The Tic can perform a homing procedure that uses a limit switch as a position reference.

To test this feature, you can open the "Device" menu in the Tic Control Center and use the "Go home reverse" or "Go home forward" command. The "Go home reverse" command moves the motor in reverse until a reverse limit switch activates, and then backs up until the limit switch deactivates. The "Go home forward" command moves the motor forward until a forward limit switch activates, and then backs up until the limit switch deactivates. The Tic Control Center software disables these commands if you have not configured limit switches in the corresponding directions. At the end of the homing procedure, after the limit switch deactivates, the Tic sets its current position to 0 and resumes normal operation. These menu options use the "Go home" command, which is supported over USB, serial, and I²C, and documented in **Section 8**.

During the homing procedure, the speed of the motor is controlled by the "Homing speed towards" and "Homing speed away" settings in the "Advanced settings" tab. The "Homing speed towards" setting controls the speed that the Tic uses while traveling towards the limit switch, while the "Home speed away" setting controls the speed that the Tic uses briefly while traveling away from the limit switch to deactivate it.

If you are using the "RC position", "Analog position", or "Encoder position" control modes, you might want to enable automatic homing so that you do not have to send the "Go home" command to the Tic. If you check the "Enable automatic homing" checkbox in the "Advanced settings" tab, the Tic will perform the homing procedure whenever it is being commanded to go to a specific position but it is uncertain about is current position (e.g. immediately after motor power is applied). The "Automatic homing direction" setting lets you choose whether the automatic homing will drive the motor in the reverse or forward direction.

5. Details

5.1. Motion parameters

This section explains the Tic settings and variables that directly control the motion of the stepper motor.

Position

The Tic represents stepper motor positions in units of microsteps, which are also called pulses. The number of microsteps that correspond to one full step is determined by the "step mode" setting.

By default, an increasing position corresponds to taking steps forward through the motor driver's current indexer table, so the amount of current flowing from B1 to B2 lags behind the amount of current flowing from A1 to A2. The "invert motor direction" setting flips this correspondence.

Positions are stored as 32-bit signed integers, so they have a range of −2,147,483,648 to +2,147,483,647 (−0x8000 0000 to +0x7FFF FFFF).

With high speeds or long system run times, it is possible for the Tic's 32-bit "current position" variable to overflow. If the Tic moves at its fastest speed (50,000 kHz) in one direction for approximately 12 hours, that would correspond to more than 2,147,483,647 microsteps. The Tic considers position +2,147,483,647 to be adjacent to position −2,147,483,648, and if you command it to go from one position to another, it will use the direction that gets it to the target position in the least number of steps, even if that involves letting the current position variable wrap around. Position overflow should not cause any issues for the Tic, but it could be important to consider it if you are writing software to control the Tic and dealing with positions.

Speed

The Tic represents speeds in units of microsteps (or pulses) per 10,000 seconds. For example, a speed of 200,000 corresponds to 20 microsteps per second. A speed is a non-negative integer that expresses how fast a stepper motor could move, but does not express direction.

The allowed range of speeds is 0 to 500,000,000 (50,000 pulses per second, or 50 kHz), but speeds from 1 to 6 will behave the same as speed 0 (see below).

Low speed considerations

Speeds from 1 to 6 will behave the same as a speed of 0 (the Tic will never take a step when the speed is so low because its step planning algorithm cannot keep track of times that long). Therefore, the minimum speed that the Tic can actually achieve is 7 steps per 10,000 seconds, or one step every 23 minutes.

It is important to note that the Tic only allows integer (whole number) speeds. So if your stepper motor

is moving at speed 10 and you want it to go 1% faster, you cannot simply change your speed to 10.1; your options would be 10 or 11. For speeds higher than 50 (1 pulse every 200 seconds), the inaccuracy caused by this is less than 1%.

High speed considerations

If you are trying to use high speeds, it is important to consider how fast your stepper motor is capable of moving. See **Section 4.3** for tips about finding your stepper motor's maximum speed.

The Tic's step planning algorithm uses a 3 MHz timer to schedule when to take steps. When the Tic is moving the motor at a certain speed, it first converts the speed into units of timer ticks by dividing 30,000,000,000 ($3×10^{10}$) by the speed, rounding down. For example, a speed of 10,000 corresponds to $3×10^{10}/1×10^4 = 3×10^6$ timer ticks, which corresponds to 1 second. The rounding from this division can cause the Tic to move slightly faster than commanded. The biggest effect is at speed 491,803,279, where steps are 60 timer ticks apart, resulting in a speed that is 1.66% faster than desired. The effect is less than 1% for any speed less than 300,000,000 (30 kHz), and less than 0.1% for any speed less than 30,000,000 (3 kHz).

Current position and current velocity

The Tic keeps track of "Current position" and "Current velocity" variables at all times. Both of these are 32-bit signed integers and they default to zero when the Tic is turned on.

The Tic uses velocity variables like "Current velocity" to hold both a speed and a direction. The absolute value of a velocity is a speed, and positive velocities correspond to increasing positions while negative velocities correspond to decreasing positions.

"Current position" can be set with a "Halt and set position" command, which has a side effect of setting the "Current velocity" to zero. The "Current velocity" can also be forced to zero by the "Halt and hold" command and by certain error conditions.

The Tic also keeps track of a flag called "Position uncertain", which indicates whether the Tic has received external confirmation that the value of its "Current position" variable is correct (see **Section 5.4**).

Target position and target velocity

Whenever the Tic is moving the stepper motor, it is either in "Target position mode" or "Target velocity mode".

In "Target position mode", the Tic has a variable called "Target position" that specifies what position the Tic is moving to. In this mode, the Tic will plan steps for the stepper motor with the goal of making its current position equal to the target position and its current velocity equal to zero, while maintaining

the speed/acceleration/deceleration limits described later in this section.

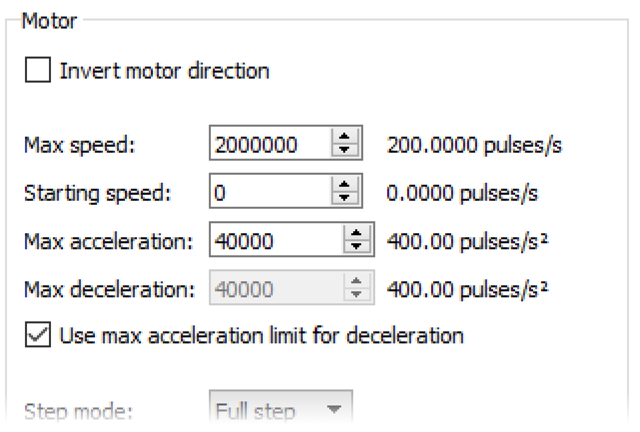
In "Target velocity mode", the Tic has a variable called "Target velocity" that specifies the velocity the Tic should maintain. It will plan steps so that its current velocity reaches the target velocity, while respecting the limits described later in this section. The target velocity can be set to any 32-bit signed integer value, but if the absolute value of the "target velocity" is larger than the max speed, then the Tic will never reach the target velocity.

In both of these modes, the Tic is designed to handle a frequently changing target. For instance, it is OK if the target position changes while the Tic is traveling towards it, and the Tic will handle changes like this seamlessly and quickly.

When the Tic's control mode is Serial / I²C / USB, you can use the "Set target position" and "Set target velocity" commands to set the target position and target velocity, respectively. These commands allow you to change between "Target position mode" and "Target velocity mode" on the fly. In the other control modes (besides STEP/DIR mode), you are limited to just one of those two modes (either target position or target velocity), and the target is set automatically based on an input to the Tic.

Max speed

The Tic's "Max speed" setting sets an upper limit on how fast the Tic will try to drive the stepper motor. See **Section 4.3** for tips about finding your stepper motor's maximum speed. The setting can be temporarily overridden using the "Set max speed" command, as described in **Section 8**.



The Tic's default motion settings.

The "Max speed" uses the Tic's standard speed units of 10,000 pulses per second, and these are the units you must use when entering it into the Tic software. For convenience, the Tic Control Center displays the maximum speed in units of pulses per second to the right its input box.

Starting speed

The Tic's "Starting speed" setting is the maximum speed at which instant acceleration and deceleration are allowed. For example, if you set the starting speed to 1,000,000 (100 pulses per second), then the Tic will be able to instantly change from any velocity in the range of −1,000,000 to +1,000,000 to any other velocity in that range. Setting the starting speed might allow you to make your system faster since it will not waste time accelerating or decelerating through low speeds where it is not needed.

This setting can be temporarily overridden using the "Set starting speed" command, as described in **Section 8**.

The "Starting speed" uses the Tic's standard speed units of 10,000 pulses per second.

Max acceleration and max deceleration

The "Max acceleration" setting specifies how rapidly the speed is allowed to increase, while the "Max deceleration" setting specifies how rapidly the speed is allowed to decrease. See **Section 4.3** for tips about choosing these settings.

These settings can be temporarily overridden using the "Set max acceleration" and "Set max deceleration" settings, as described in **Section 8**. However, note that the "Set max acceleration" command never changes the "Max deceleration" value, even if the "Use max acceleration limit for deceleration" checkbox is checked.

Both of these settings use units of pulses per second per 100 seconds. In other words, they specify how much the speed can rise or fall in one hundredth of a second (0.01 s or 10 ms). The range of allowed values is 100 to 2,147,483,647.

5.2. Analog/RC input handling

This section documents the details of how the Tic reads its analog and RC inputs and how it averages, filters, and scales those inputs in order to set a target velocity or target position.

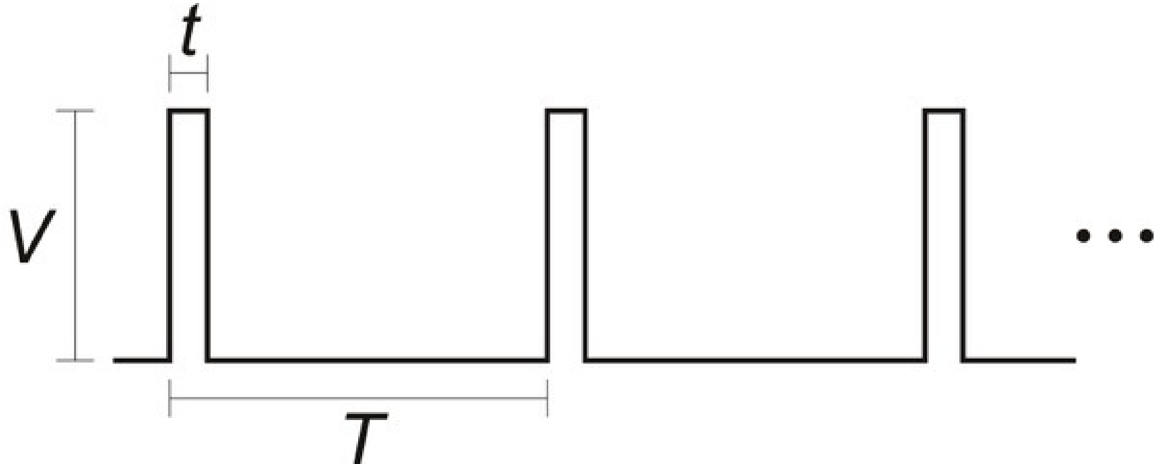
Analog readings on the SDA pin

When the SDA pin is configured as an analog input, the Tic regularly uses its 10-bit ADC to take readings of the voltage on the pin. After it completes 8 ADC readings, it adds the readings together and multiplies by 8 to get a number between 0 and 65,472 which it stores in the "Analog reading SDA" variable. The "Analog reading SDA" variable, like most of the variables in this section, can be read from the Tic over USB, serial, or I²C, and is listed in **Section 7**.

Pulse measurement on the RC pin

The Tic measures the width of RC pulses on its RC input pin. The signal on the RC pin should look like the waveform shown below:

© 2001–2019 Pololu Corporation



The signal should be low (0 V) normally and have periodic pulses with an amplitude (*V*) of at least 2 V. The width of the pulses (*t*) should be between 200 μs and 2700 μs. The period of the signal (*T*), should be at most 100 ms, and the there is no particular lower limit. When the Tic has received three good pulses in a row, it writes the width of the latest pulse to the "RC pulse width" variable in units of 1/12 μs.

If the Tic goes more than 100 ms without receiving a good pulse, it will change the "RC pulse width" variable to 0xFFFF (65,535) to indicate that the RC signal has been lost. Furthermore, if the Tic goes more than 500 ms without receiving three good pulses in a row, it will change the "RC pulse width" variable to 0xFFFF to indicate the RC signal is unreliable.

Input averaging

The Tic's input averaging feature, which is enabled by default, helps smooth out noisy RC or analog input signals by taking a running average.

By default, when the Tic's control mode is set to any of the RC or analog options, the Tic calculates a running average of the last four readings (from the "Analog reading SDA" or "RC pulse width" variables) and stores that average (possibly scaled by a power of two) in the "Input after averaging" variable. If you disable input averaging by unchecking the "Enable input averaging" checkbox in the Tic Control Center, then the "Input after averaging" variable is set directly from the corresponding input variable without averaging.

Input hysteresis

Another optional feature that helps deal with noisy inputs is input hysteresis. The input hysteresis feature can be useful if you are using a noisy input to set the position of a stepper motor, and you want the stepper motor to be at rest whenever you are not changing the input, instead of moving around by small amounts and making noise.

This feature is turned off by default, but can be turned on by entering a value of 2 or more into the "Input hysteresis" setting in the "Input conditioning" box of Tic Control Center. When the hysteresis value is 2 or more, the Tic essentially takes the "Input after averaging" variable and maps it to the nearest number that is a multiple of the "Input hysteresis" value. However, it does this mapping with hysteresis; the result of this process, which is stored in the "Input after hysteresis" variable, only changes if the input is at least one hysteresis value away from the current result.

For example, suppose your hysteresis value is 100, and your "Input after averaging" starts out at 670. The initial "Input after hysteresis" value will be 600 (it always rounds down initially; afterward, it always rounds toward the current "Input after hysteresis" value). The "Input after hysteresis" will change to 500 if the "Input after averaging" drops to a value between 401 and 500, and it will change to 700 if the "Input after averaging" rises to a value between 700 and 799. If the "Input after averaging" remains between 501 and 699, the "Input after hysteresis" will not change.

When the input hysteresis feature is disabled, the "Input after hysteresis" variable is just a copy of "Input after averaging".

One way to choose a good hysteresis value would be look at the "Input after averaging" variable that is displayed in the Tic Control Center, see how much it varies when you are not moving the input, and set the hysteresis value to something larger than that.

Input before scaling

The Tic's input scaling routine requires a number between 0 and 4095, but the "Input after hysteresis" variable might be more than that. The "Input after hysteresis" variable is divided by a power of two to make a variable called "Input before scaling" that is between 0 and 4095.

The "Input before scaling" variable cannot be directly read from the Tic but it can be computed from the "Input after hysteresis" variable using the units table below. For example, if you are in an analog control mode with input averaging enabled, you can get the "Input before scaling" by reading the "Input after hysteresis" variable and dividing by 8. If you are in an analog control mode without input averaging enabled, you would only divide by 2. If you are in an RC control mode, you would divide by 8.

Input variable units

The table below specifies the units of the input variables defined above. In analog mode, the units are specified by how many bits the reading has. For example, a 12-bit reading would be a number between 0 to 4095, where 0 represents 0 V and 4095 represents the Tic's logic voltage (the voltage on the 5V pin).

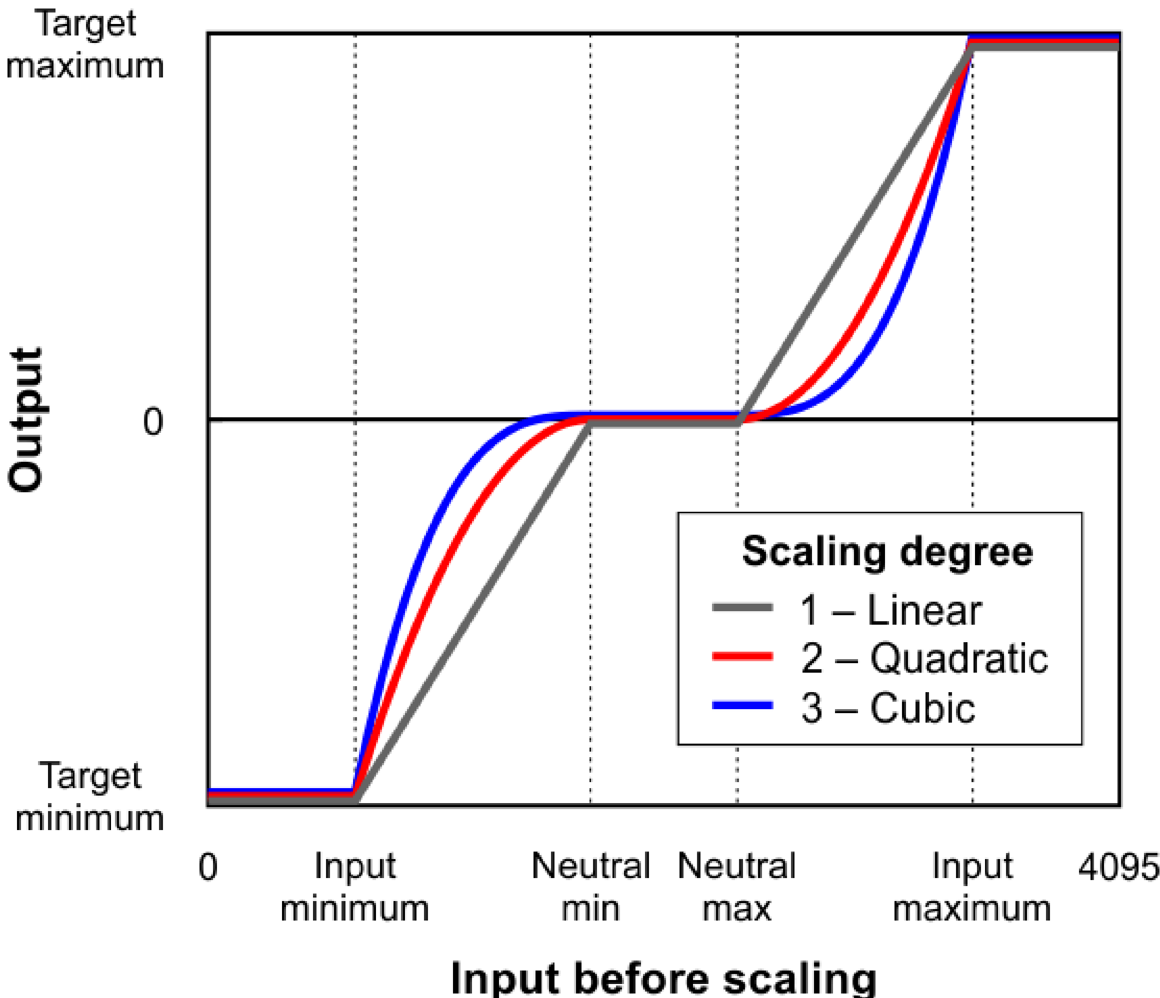
Control mode	Input averaging enabled	Input after averaging, Input after hysteresis, Input hysteresis	Input before scaling
Analog	Yes	15-bit	12-bit
Analog	No	13-bit	12-bit
RC	Yes	1/12 μs	2/3 μs
RC	No	1/12 μs	2/3 μs

RC and analog scaling

The settings in the “RC and analog scaling” box of the Tic Control Center determine how the “Input before scaling” variable gets mapped to a signed 32-bit integer that will either be used as a target position or a target velocity depending on the Tic’s control mode. For tips about how to set up these settings when you are getting started with the controller, see the section for your control mode under **Section 4**.

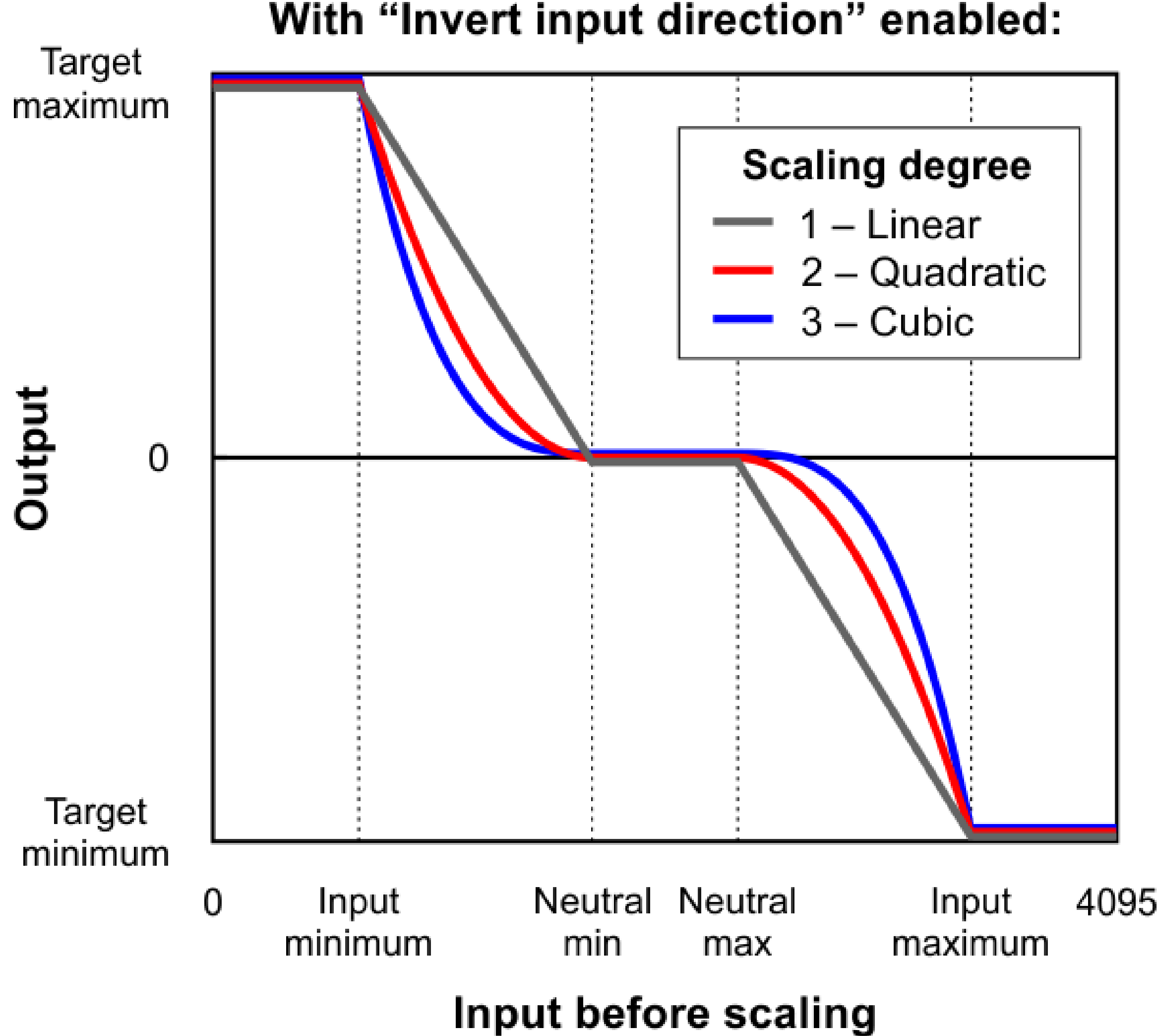
The graphs below illustrate how input values are mapped to target values:

© 2001–2019 Pololu Corporation



This graph shows how the Tic's RC/analog input is scaled to produce a target position or target velocity (input direction not inverted).

© 2001–2019 Pololu Corporation

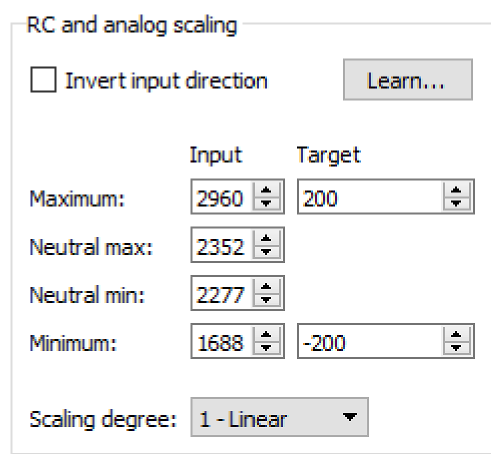


This graph shows how the Tic's RC/analog input is scaled to produce a target position or target velocity (input direction inverted).

When the "Invert input direction" checkbox is not checked, the input values are mapped to output/ target values according to these rules:

© 2001–2019 Pololu Corporation

- Any input value greater than the input maximum gets mapped to the target maximum.
- Any input value between the input neutral max and the input maximum gets mapped to a number between 0 and the target maximum.
- Any input value between the input neutral min and input neutral max gets mapped to 0.
- Any input value between the input minimum and input neutral min gets mapped to a number between the target minimum and 0.
- Any input value less than the input minimum gets mapped to the target minimum.



Example Tic scaling settings for RC position control mode.

When the "Invert input direction" checkbox is checked, it changes the scaling so that higher input values correspond to lower target values. You can think of it as simply switching the target maximum and target minimum in the rules above.

The scaling degree can be set to "1 – Linear", "2 – Quadratic", or "3 – Cubic". With the default setting of "1 – Linear", the scaling function is linear. If you choose a higher scaling degree, then the Tic uses a higher-degree polynomial function, which can give you finer control when the input is closer to its neutral position. With linear scaling, if the input is one quarter (1/4) of the way from the input neutral max to the input maximum, the target will be one quarter (1/4) of the target maximum. With quadratic scaling, the output would be one sixteenth (1/16) of the target maximum. With cubic scaling, the output would be one sixty fourth (1/64) of the target maximum.

The input maximum, input neutral max, input neutral min, and input minimum must be between 0 and 4095. The target maximum must be between 0 and +2,147,483,647, while the target minimum must be between 0 and -2,147,483,647.

Analog/RC input state

When the Tic's control mode is analog or RC, the Tic's "Input state" variable starts off in the "not ready" state when the Tic starts running and has not yet collected its first set of analog readings or detected the RC signal. The Tic leaves this state quickly (usually within 100 ms). After that, the Tic's input state will be in "invalid" if its RC input signal is missing or bad. There is currently no concept of an analog signal being invalid, so the analog control modes do not use that state. If the analog/RC input is valid, then the input state variable will either be "Target position" or "Target velocity" depending on what control mode was selected. This indicates that the Tic has successfully read its input and populated the "Input after scaling" variable with either a target position or a target velocity. For details about how

© 2001–2019 Pololu Corporation

the "Input state" variable is encoded, see **Section 7**.

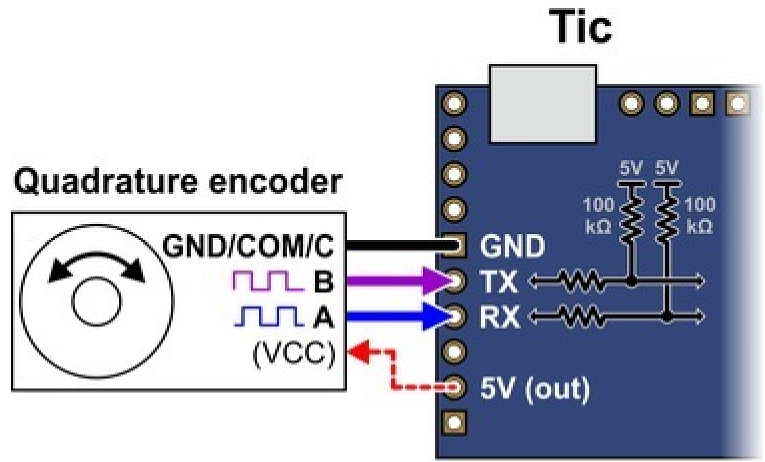
5.3. Encoder input handling

This section explains the details of how the Tic reads quadrature encoder signals and converts them to a target position or velocity.

The Tic does **not** support closed-loop control with encoder feedback. The Tic's encoder input is meant to be connected to a rotary encoder that is turned by hand to control the stepper motor's position or speed. If you have a stepper motor with an integrated encoder, you should not try to connect the motor's encoder to the Tic.

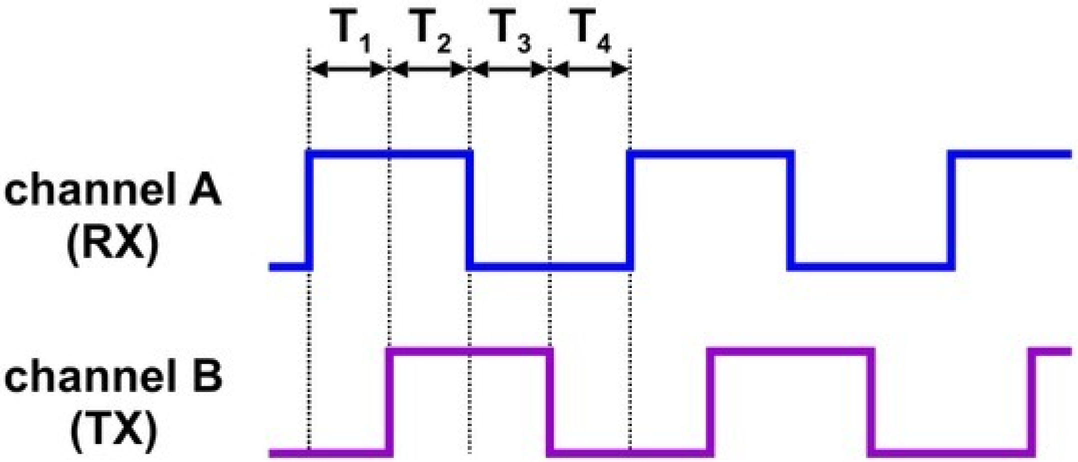
Encoder monitoring on the RX and TX pins

The Tic's RX and TX pins are normally used for TTL serial communication, but they can be configured as encoder inputs, and that is their default configuration when the Tic's control mode is set to "Encoder speed" or "Encoder position". Each of these pins has a 100kΩ pull-up resistor and a series resistor, as shown in the diagram below.



The Tic expects to see a quadrature encoder signal like this on th RX and TX pins:

© 2001–2019 Pololu Corporation



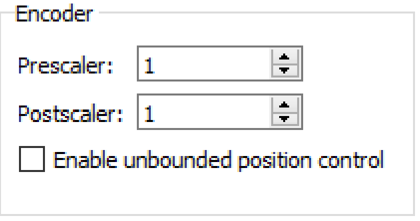
The time between the channel transitions (labeled $T_{1\text{-}4}$ in the diagram above) must be at least 100 μs. The signal must rise to 4 V or higher, and must stay within 0 V to 5 V.

The Tic stores the raw encoder counts in the "Encoder position" variable. This variable can be read over serial, I²C, or USB, and it is displayed in the "Status" tab of the Tic control center. The "Encoder position" variable gets reset to zero when the Tic's motor power supply is too low, when the Tic is experiencing a motor driver error, when the Tic's control mode is changed, and when the Tic receives a Reset command.

An increasing encoder position corresponds to the TX signal leading the RX signal, and a decreasing encoder position corresponds to the RX signal leading the TX signal.

Encoder scaling

When the Tic's control mode is set to "Encoder position" or "Encoder speed", the encoder prescaler and postscaler settings, along with the "Enable unbounded position control" option, determine how the encoder position maps to the "Input after scaling" variable, a 32-bit signed integer which is used to set the target position or target velocity of the motor.



The Tic's default encoder scaling settings.

The Tic internally keeps track of an encoder "cursor" variable, which is always a multiple of the encoder prescaler. When the encoder position is more than the prescaler value away from the cursor, the cursor increases or decreases by the prescaler amount to get closer to the encoder position. When the cursor increases or decreases, the Tic adds or subtracts the postscaler value to the "Input after scaling" variable if unbounded position control is enabled *or* the change to the "Input after scaling" variable would keep it within the bounds defined by the "Target maximum" and "Target minimum" settings.

The unbounded position control option can only be enabled when the control mode is “Encoder position”, not “Encoder speed”. With this option enabled, the “Input after scaling” can overflow and wrap around from +2,147,483,647 to −2,147,483,648.

When the Tic’s control mode is “Encoder position”, the Tic sets it target position equal to the “Input after scaling” variable described above. For tips about how to set the encoder scaling parameters in encoder position mode, see **Section 4.11**.

When the Tic’s control mode is “Encoder speed”, the Tic sets its target velocity equal to the “Input after scaling” variable described above. For tips about how to set the encoder scaling parameters in encoder speed mode, see **Section 4.12**.

5.4. Error handling

This section explains the details of how the Tic detects and handles error conditions. The table below summarizes the Tic’s error handling:

Operation state	Conditions	Effects
Reset	• **Motor driver error** • **Low VIN**	• **De-energize** • **Drive ERR line high** • **Reset driver** • **Clear encoder count**
De-energized	• **Intentionally de-energized**	• **De-energize** • **Drive ERR line high**
Soft error	• **Kill switch active** • **Required input invalid** • **Serial error** • **Command timeout** • **Safe start violation**	• **Drive ERR line high** • **Soft error response**
Waiting for ERR line	• **ERR line high**	• **Soft error response**
Starting up	• **RC/analog input not ready** • **Coil current stabilizing**	• **Energize**
Normal	• (None of the above)	• **Energize** • **Obey input** • **Learn position**

The Tic constantly evaluates the conditions listed above to see if they are happening or not. The detailed definitions of these conditions are explained below. If a condition is happening, the Tic will

go into the corresponding operation state as shown in the table above. If multiple conditions are happening, the Tic will choose the *first* state (the highest state in the table above) that has an active condition. If no conditions are happening, the Tic will proceed with normal operation. After the Tic determines what operation state it is in, it performs the corresponding effects for that operation state as listed in the table. These effects are explained in detail below.

Error handling variables

The “Operation state” variable says what operation state the Tic is currently in. The “Error status” variable says what error conditions are currently happening: it has bits for most of the conditions in the table above. The “Errors occurred” variable helps detect errors that only occur for a brief period of time: it contains bits that get set whenever an error happens, and can be cleared with a command. These variables can be read over TTL serial, I²C, or USB. For more information about these variables, see **Section 7**.

Condition: Motor driver error

A motor driver error means that the motor driver IC on the Tic has detected a problem and reported it to the main microcontroller.

When the Tic detects a motor driver error, it latches the error, meaning that the “Motor driver error” bit in the “Error status” variable will stay set even after the motor driver stops signalling the fault condition. By default, the Tic will clear the latched error every 0.5 s, but you can disable this behavior by unchecking the “Automatically clear driver errors” checkbox in the Tic Control Center. The Tic will also clear the latched error whenever it receives a “Clear driver error” command.

The MP6500 motor driver IC on the Tic T500 reports motor driver errors due to over-current, over-temperature, and over-voltage conditions.

The DRV8825 motor driver IC on the Tic T825 reports motor driver errors due to over-current and over-temperature conditions.

The DRV8834 motor driver IC on the Tic T834 reports motor driver errors due to over-current, over-temperature, and under-voltage conditions.

The TB67S249FTG motor driver IC on the Tic T249 reports motor driver errors due to over-current, over-temperature, and open-load conditions. However, the Tic ignores the open-load fault because it happens frequently during normal operation.

When the Tic T249 detects a motor driver error, it reports the type of error in the “Last motor driver error” variable, which can be read via serial, USB, and I²C, and is displayed in the “Status” tab of the Tic control center. This feature is not available on the other Tics.

In response to a motor driver error, the Tic will go into the Reset state and assert the motor driver's reset line. In many cases, resetting the motor driver causes it to stop reporting the error. The Tic latches the motor driver error so that it can avoid immediately going back to its previous state when the driver stops reporting the error and causing a fast oscillation between states.

Motor driver errors are ignored during the brief time when the **coil current is stabilizing** because the Tic T834 reports a motor driver error for a few milliseconds after coming out of its Reset state. The Tic needs to ignore that error so it can successfully enable the motor the driver and not go back into the Reset state.

Condition: Low VIN

The "Low VIN" error indicates that the voltage on VIN has dropped well below the minimum operating voltage.

On the Tic T500, the minimum operating voltage is 4.5 V, and the threshold for the "Low VIN" error is fixed at **3.0 V**.

On the Tic T825, the minimum operating voltage is 8.5 V, and the threshold for the "Low VIN" error is fixed at **7.0 V**.

On the Tic T834, the minimum operating voltage is 2.5 V, and the threshold for the "Low VIN" error is fixed at **2.1 V**.

On the Tic T249, the minimum operating voltage is 10 V, and the threshold for the "Low VIN" error is fixed at **5.5 V**.

Condition: Intentionally de-energized

The "Intentionally de-energized" error bit is 0 when the Tic starts up. It can be set with the "De-energize" command and cleared with the "Energize" command. Those two commands are used to implement the "De-energize" and "Resume" buttons in the Tic Control Center (the "Resume" button also issues an "Exit safe start" command).

You can use those commands over serial, I²C, or USB to turn off your motor and reduce power consumption. However, note that if you do this, then the "De-energize" button in the Tic Control Center will not be as useful because its effect will get undone when your device issues an "Energize" command.

Condition: Kill switch active

The "Kill switch active" error indicates that one of the pins configured as a kill switch is in its active state. For more information about kill switches, see **Section 5.5**.

Condition: Required input invalid

This error indicates that the Tic's main input signal is not valid, so it cannot be used to set the target position or velocity. This error currently only happens in RC control modes.

Condition: Serial error

This error indicates that something went wrong with the I²C or TTL serial communication. When the control mode is "Serial / I²C / USB", this bit gets set whenever any of the following conditions happen:

- The Tic receives an invalid command byte or data byte over I²C or TTL serial. (This also causes the "Serial format" bit in the "Errors occurred" variable to be set.)
- The Tic receives a TTL serial byte with its most significant bit set (which starts a new command) while it is still waiting for data bytes from the previous command. (This also causes the "Serial format" bit in the "Errors occurred" variable to be set.)
- The Tic receives an incorrect CRC byte at the end of a command. (This also causes the "Serial CRC" bit in the "Errors occurred" variable to be set.)
- The Tic receives a TTL serial byte at a time when its hardware or software buffers are not able to hold the byte, and the byte is lost. (This also causes the "Serial RX overrun" bit in the "Errors occurred" variable to be set.) This should not happen under normal conditions.
- The Tic receives a TTL serial byte that does not have a valid stop bit. (This also causes the "Serial framing" bit in the "Errors occurred" variable to be set.)

In a non-serial control mode, serial errors do not cause the Tic to shut down because the serial error will never get set. However, the serial-related bits in the "Errors occurred" register will still be set, so you can detect serial errors in non-serial control modes.

Starting in firmware version 1.06, the Tic ignores the invalid command bytes 0xFF and 0xFE without triggering an error if they are received via TTL serial. This helps make the Tic less susceptible to noise on its serial lines.

The serial error is cleared whenever the Tic receives any of following commands over serial, I²C, or USB:

- Set target position
- Set target velocity
- Halt and set position
- Halt and hold
- Energize

- Exit safe start
- Reset

Condition: Command timeout

The Tic keeps track of how much time has passed since it last received a serial, I²C, or USB command that is one of the commands that clears the serial error (listed above), or the “Reset command timeout” command. Whenever the Tic’s control mode is “Serial / I²C / USB” and that time exceeds the timeout period, which is one second by default, the Tic sets the “Command timeout” error bit. You can change the timeout period or disable this feature from the “Input and motor settings” tab of the Tic Control Center. When the timeout error occurs, you can clear it by sending one of the commands that clears the serial error (listed above) or by sending the “Reset command timeout” command.

The Tic Control Center sends the “Reset command timeout” command repeatedly while it is connected to a Tic over USB.

The “Command timeout” error is useful if you want to control the Tic over serial, I²C, or USB, and you want to be sure that the stepper motor will stop moving if the device controlling the Tic stops working.

Condition: Safe start violation

The Tic’s safe start feature helps to avoid unexpectedly powering the motor in speed control modes and in “Serial / I²C / USB” mode.

In the “RC *speed*”, “Analog *speed*”, or “Encoder *speed*” control mode, the Tic will generally set the “Safe start violation” error bit whenever the Tic’s “Operation state” is not “Normal” and the “Input after scaling” variable (which represents the target velocity being commanded by the Tic’s main input source) is non-zero. This way, the Tic will never automatically start moving the motor just because all the error conditions have been resolved; it will only start moving the motor if you first put its main input in the neutral position (which maps to a speed of 0), and then move its main input away from the neutral position. If an error happens while you are operating the motor, you will need to move your main input back to the neutral position before you can start moving the motor again.

In the “Serial / I²C / USB” control mode, the Tic will generally set the “Safe start violation” error bit whenever the Tic is not in the normal operation state. You can send an “Exit safe start” command to clear the error for 200 milliseconds, giving the Tic a chance to start up. In general, you should only send the “Exit safe start” command as a direct response to a user action, such as pressing a button. That way, if an error happens and your motor stops, it will only start moving again in response to the user’s action.

You can send the “Enter safe start” command to get back into safe start mode. This command forces the “Safe start violation” bit to be set the next time the Tic evaluates it. After that, the “Safe start

violation" bit will only be cleared if the usual conditions for clearing it are satisfied, as described above. In position control modes, where the safe-start feature generally has no effect, this command will cause a temporary "Safe start violation" error that could interrupt movement of the motor.

You can check the "Disable safe start" checkbox in the "Advanced settings" tab of the Tic Control Center to make the "Safe start violation" error never happen, even if an "Enter safe start" command is sent.

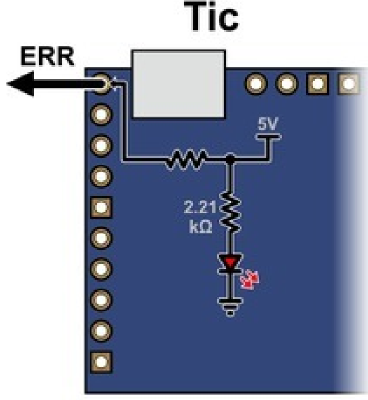
> The safe-start feature does not do anything useful in the "RC *position*", "Analog *position*", or "Encoder *position*" control modes. In those modes, some safety can be provided by the "Learn position" feature as described in the **Effect: Learn position** section below.

Condition: ERR line high

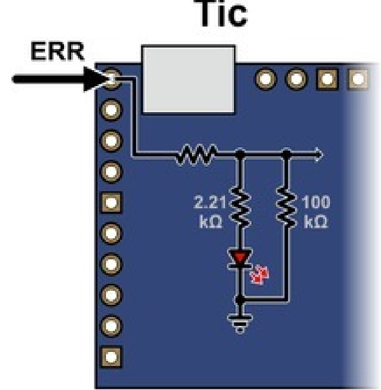
The Tic reports an "ERR line high" error if it is not driving its ERR pin high and the digital reading on the ERR pin input is high. The error is cleared automatically when either of those conditions stops being true. You can use the "Ignore ERR line high" setting to disable the error.

The Tic uses its ERR pin, which is connected to the red LED, to report errors. Whenever the Tic drives the ERR line high to indicate an error, the red LED turns on. The red LED is in series with a 2.21kΩ resistor. There is a 470Ω resistor with the ERR pin on one side, and the Tic's microcontroller pin and red LED on the other side.

When there is no error happening, the Tic's ERR line acts as a digital input so it can detect error signals from other devices. It is pulled low by the LED and a 100 kΩ pull-down resistor. The voltage on the Tic's microcontroller pin must reach at least 2 V for the input to be guaranteed to read as high.



Schematic diagram of the Tic ERR pin when it is acting as an output (i.e. there are errors).



Schematic diagram of the Tic ERR pin when it is acting as an input (i.e. there are no errors).

You can connect the ERR lines of multiple Tics together so that when one of the Tics experiences an error and drives its ERR line high, all of the Tics will be notified about the error and shut down. Each Tic you add to the system will load down the ERR lines of the other Tics, so there is a limit to how many Tics you can connect in this way. It should work fine with 5 or fewer Tics. You can probably connect more than that, but at some point, there would be too much load for a single Tic to shut down all of the others. If you want to connect many Tic ERR lines together and find this to be a problem, you could desolder or disconnect the red LED on several of the Tics to reduce that load.

Condition: RC/analog input not ready

The Tic cannot instantly determine the value of its RC or analog inputs: it takes some time for the Tic to get readings and compute averages. In RC and analog control modes, the "RC/analog input not ready" condition prevents the Tic from going into normal operation until its input is ready. This is a temporary condition that should last for at most 500 ms. It only happens right after the Tic starts up, receives a Reset command, or changes its control mode.

Condition: Coil current stabilizing

This condition prevents the Tic from going into normal operation within 10 milliseconds after it changes from the special current limit during error back to the normal current limit, or within 10 milliseconds of resetting the stepper driver.

Effect: De-energize

When the Tic is in the "Reset" or "De-energized" state, it will disable the driver (by driving its $\overline{\text{ENBL}}$ pin high) and stop sending step pulses to the driver. The current flowing through the motor coils will stop.

Effect: Drive ERR line high

When the Tic is experiencing any error (except for the "ERR line high" error), the Tic will drive its ERR line high, which causes its red LED to turn on.

Effect: Reset driver

In the "Reset" operation state, the Tic will reset its stepper motor driver IC. This means that its indexer will be reset to its default position in the table of coil currents.

Effect: Clear encoder count

In the "Reset" state, the Tic constantly sets the "Encoder position" variable, which holds the raw measurement of encoder counts, to 0.

Effect: Soft error response

When the Tic goes from the normal operation state to either the "Soft error" state or the "Waiting for ERR line" state, it will do one of four things depending on how it is configured. You can select one of

the following responses from the "Soft error responses" box in the "Advanced settings" tab of the Tic Control Center:

- **De-energize:** The Tic will de-energize the motor and set the "Position uncertain" flag.
- **Halt and hold:** The Tic will abruptly stop taking steps, keep the motor energized, and set the "Position uncertain" flag.
- **Decelerate to hold:** The Tic will set its "Target velocity" to 0, causing the motor to decelerate to a stop and then hold its position. This is the default setting.
- **Go to position:** The Tic will set its "Target position" to the specified position.

Note that the effects above only happen when the Tic's "Operation state" changes from "Normal" to "Soft error" or "Waiting for ERR line". These effects do not happen if there is a soft error right after the Tic starts up, receives a Reset command, or changes its control mode.

You can also set a special current limit to be used while there is a soft error. If enabled, this current limit will be in effect whenever the Tic is in the "Soft error" or "Waiting for ERR line" states, regardless of how it got to those states. To set this current limit, check the "Use different current limit during soft error" checkbox and enter the current limit. The special current limit will only take effect after the Tic stops moving the motor.

Effect: Energize

During normal operation or while it is starting up, the Tic will energize the stepper motor coils by enabling its stepper motor driver.

Effect: Obey input

During normal operation, the Tic sets the stepper motor's "Target position" or "Target velocity" as specified by its main input.

The motion specified by the main input is held in the "Input state" and "Input after scaling" variables. In the "Serial / I²C / USB" control mode, those variables are set by the "Set target position", "Set target velocity", "Halt and hold", "Halt and set position", and "Reset" commands, and they do not get reset when an error happens. In other control modes, those variables are set automatically.

Effect: Learn position

When the control mode is "RC position", "Analog position", or "Encoder position", and the Tic starts normal operation, it will check the "Position uncertain" flag. If that flag is set, the Tic will assume that the stepper motor has already reached the position specified by the main input, and so it will set its "Current position" variable equal to the "Target position", and clear the "Position uncertain" flag.

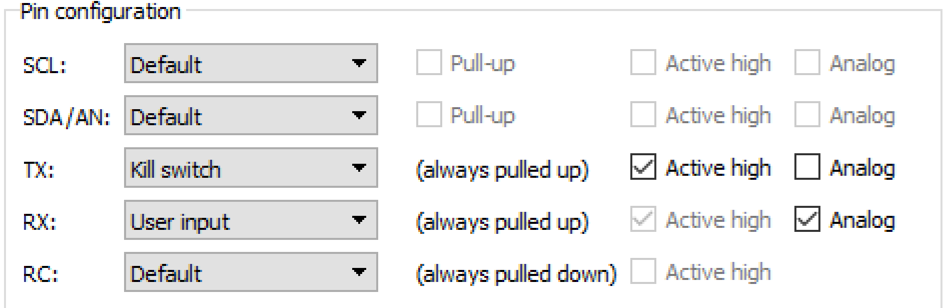
This means that when you power on your system, or recover from certain errors, the Tic will assume that its position is already correct, and not start moving right away to reach a new position. This feature is similar to the safe start feature described above for the other control modes, because it helps prevent the motor from moving by surprise.

If the Tic's "Soft error response" is set to "Decelerate to hold" (the default) or "Go to position", then the "Position uncertain" flag will **not** be set when the Tic has a soft error, and therefore the Tic will not learn its position when the soft error is resolved, and this could cause unexpected motion. Setting the Tic's "Soft error response" to "De-energize" or "Halt and hold" makes unexpected motion less likely.

5.5. Pin configuration

This section explains how to configure the control pins of the Tic: SCL, SDA/AN, TX, RX, and RC.

The settings for these pins are found in the "Pin configuration" section of the "Advanced settings" tab in the Tic Control Center, as shown below:



Example pin configuration settings for a Tic Stepper Motor Controller.

To the right of each pin name is a drop-down box where you can assign a function to the pin. Each pin function is documented below.

Pin function: Default

By default, each pin's function is set to "Default". This means that the Tic will automatically choose the pin's function and choose the related options (Pull-up, Active high, Analog) based on the "Control mode", as shown in the table below.

Control mode	RC position, RC speed, Serial / I²C / USB, or STEP/DIR	Analog position, Analog speed	Encoder position, Encoder speed
Default function for SCL	Serial (SCL), pull-up enabled	Potentiometer power	Serial (SCL), pull-up enabled
Default function for SDA/AN	Serial (SDA), pull-up enabled	User input, analog enabled	Serial (SDA), pull-up enabled
Default function for TX	Serial (TX)	Serial (TX)	Encoder input
Default function for RX	Serial (RX)	Serial (RX)	Encoder input
Default function for RC	RC input	RC input	RC input

Note that the TX and RX pins are always pulled up, and the RC pin is always pulled down.

When a pin's function is "Default", the Tic Control Center disables the checkboxes for the Pull-up, Active high, and Analog options because they will have no effect. However, the states of those checkboxes will still be saved to the Tic.

Pin function: User input

A pin with the "User input" function is used as a digital and/or analog input. You can read the state of the input using the "Get variable" command or by running `ticcmd --status --full`. See the **Variable reference section** [https://www.pololu.com/docs/0J71/7] for more information about the user input variables.

Pin function: User I/O

For now, the "User I/O" function is exactly the same as "User input". (In the future, we might add commands allowing you to turn User I/O pins into outputs and drive them high or low.) This function is only available for the SCL, SDA, TX, and RX pins. The RC pin cannot be an output due to hardware limitations.

Pin function: Potentiometer power

A pin with the "Potentiometer power" function drives high so that it can be used to power a potentiometer for the Tic's analog input. This function is only available for the SCL pin.

Pin function: Serial

A pin with the “Serial” function acts as the TTL serial or I²C pin as described by its name. This function is only available for the SCL, SDA, TX, and RX pins. It is possible to enable the RX pin as a serial pin without enabling TX if you just want to send commands to the Tic without reading anything back. (It is also possible to enable TX as a serial pin without enabling RX, but this is not useful since the Tic only sends data on TX in response to commands received on RX.) It is not possible to enable the SCL or SDA pin as a serial pin without enabling both, so the Tic Control Center will warn you and offer to fix your settings if try to do that.

Pin function: RC input

A pin with the “RC input” function is used to measure incoming RC pulses. This function is only available on the RC pin, and configuring the RC pin as a “User input” actually has the same effect because the Tic’s RC pulse measuring system is always active, regardless of what the RC pin is actually being used for.

Pin function: Encoder input

A pin with the “encoder input” function is used to read signals from a quadrature encoder. This function is only available on the TX and RX pins, and it is actually the same as “User input” on these pins because the Tic’s encoder reading system is always active, regardless of what the TX and RX pins are actually being used for.

Pin function: Kill switch

A pin with the “Kill switch” function is used as a digital input that can tell the Tic to stop moving the motor. If the “Active high” checkbox is checked, that means that the kill switch is considered active whenever its digital reading is high (5 V). If the “Active high” checkbox is not checked, the switch is considered active whenever its digital reading is low (0 V). The Tic sets its “Kill switch active” error bit whenever any of the pins configured as kill switches are in their active state, and it clears the bit once all of the kill switches have left the active state.

As described in **Section 5.4**, the kill switch error is considered to be a “Soft error”. By default, the Tic will decelerate to zero speed and hold position when a kill switch is triggered, but you can change this behavior using the “Soft error response” setting.

Pin function: Limit switch forward/reverse

A pin with the “Limit switch forward” function is used as a digital input that can tell the Tic to prevent the motor from going in the forward direction. Similarly a pin with the “Limit switch forward” function can tell the Tic to prevent the motor from going in the reverse direction. The “Active high” checkbox controls the polarity of the limit switch input.

If a limit switch is active while the motor is moving in the direction limited by the switch, the Tic abruptly halts motor movement and sets the “Position uncertain” flag.

See **Section 4.14** for information about setting up limit switches.

Pin option: Pull-up

The “Pull-up” checkbox allows you to enable the internal pull-up resistor for a pin. This option is only available for the SCL and SDA pins. The TX and RX pins are always pulled up with 100 kΩ resistors which cannot be disabled. The RC pin input is always pulled down.

If the “Pull-up” checkbox is disabled (grayed out), it means that the value of the checkbox will have no effect on the Tic, either because the pin function is “Default”, so the pull-up option is based on the control mode of the device as described above, or because a pin function is selected that never uses pull-ups (e.g. potentiometer power).

Pin option: Active high

For pins configured as kill switch inputs, the “Active high” checkbox lets you choose whether the switch is active when the digital reading is high (checked) or low (unchecked).

If the “Active high” checkbox is disabled (grayed out), it means that the value of the checkbox will have no effect on the Tic because the pin’s function is not set to “Kill switch”.

Pin option: Analog

The “Analog” checkbox tells the Tic to do analog readings on the specified pin. The RC pin is not capable of doing analog readings due to hardware limitations.

If the “Analog” checkbox is disabled (grayed out), it means that the value of the checkbox will have no effect on the Tic because the pin function is “Default”. In that case, the analog option is based on the control mode of the device as described above.

You can read the state of the analog inputs using the “Get variable” command. See the **Variable reference section** **[https://www.pololu.com/docs/0J71/7]** for more information about the analog reading variables.

Because the SDA/AN pin is used to control the speed or position of the stepper motor in the “Analog position” and “Analog speed” modes, it gets special treatment whenever its analog input is enabled. For the SDA/AN pin, the Tic averages together eight 10-bit ADC readings, whereas the Tic just does a single 10-bit ADC reading at a time for the other pins.

5.6. Homing

This section documents the details of the Tic's homing procedure. For information about setting up limit switches and homing, see **Section 4.14**.

The homing procedure can be started using the "Go home" command documented in **Section 6** or the automatic homing feature described in **Section 6**. When you start the homing procedure, you must specify a direction: forward or reverse.

The homing procedure performed by the Tic consists of the following steps:

1. Set the "Position uncertain" flag.
2. Set the target velocity in order to move the stepper motor towards a limit switch. The sign of the velocity is determined by the specified homing direction: forward homing corresponds to a positive velocity and reverse homing corresponds to a negative velocity. The magnitude of the velocity is specified by the "Homing speed towards" setting.
3. Wait until a limit switch corresponding to the homing direction is active. For forward homing, this means that the Tic waits for a forward limit switch to be active. For reverse homing, the Tic waits for a reverse limit switch to be active. If you have not configured or wired your limit switches properly, this step might run indefinitely. When the limit switch is active, it causes the motor to stop abruptly, overriding the target velocity set in the previous step.
4. Wait for 20 milliseconds.
5. Set the target velocity in order to move the stepper motor away from the limit switch. The sign of the velocity is determined by the specified homing direction, and is the opposite of the sign chosen in step 2. The magnitude of the velocity is specified by the "Homing speed away" setting.
6. Wait for the limit switch corresponding to the homing direction to deactivate.
7. Halt the motor and set the current position to 0.
8. Clear the "Position uncertain" flag.
9. If the control mode is "Serial / I²C / USB", command the motor to halt, disregarding other motion commands that were previously received.

The homing procedure is aborted early if the Tic receives any of these commands:

- De-energize
- Reset

The homing procedure is aborted early if the Tic's control mode is set to "Serial / I²C / USB" and it

receives any of the following commands:

- Set target position
- Set target velocity
- Halt and set position
- Halt and hold

The homing procedure is also aborted early if the Tic is in any operation state other than "Normal" (i.e. if an error happens), if the Tic loses power, or if the Tic is reset using the $\overline{\text{RST}}$ line.

You can tell whether the Tic is running the homing procedure by reading the "Homing active" variable documented in **Section 7**.

5.7. Upgrading firmware

The Tic has field-upgradeable firmware that can be easily updated when Pololu releases bug fixes or new features.

Firmware versions

- **Version 1.00**, released 2017-07-14: This is the original version.
- **Version 1.01**, released 2017-07-20: This version fixes a bug where the encoder input would not work after the Tic leaves USB suspend mode.
- **Version 1.02**, released 2017-08-02: This version fixes an issue with the I²C interface where it could stop working if there was any activity on the bus within 2 ms after the end of a read. It also adds special LED blinking on startup for abnormal resets, and adds support for the "Serial response delay" setting.
- **Version 1.03**, released 2017-11-02: This is the first version with support for the Tic T834. Starting with this version, motor driver errors are latched. This version fixes two bugs that had a small chance of causing watchdog resets when the Tic was configured improperly, experiencing an error, or applying new settings. This version fixes a bug that caused the Tic connected to a suspended USB bus to stay in sleep mode even after VIN power is connected. This version fixes a bug where the motor would not move in encoder control modes if the "Invert input direction box" was checked and the "Enable unbounded position control" checkbox was not checked. As of this version, an invalid command byte received over I²C now results in the "Serial format" bit in the "Errors occurred" register being set. It also disables blinking of the green LED during suspend mode.
- **Version 1.04**, released 2018-03-14: This is the first version with support for the Tic T500. It also restores blinking of the green LED in suspend mode.

- **Version 1.06**, released 2019-01-31: This is the first version with support for the Tic T249. It also adds support for limit switches, the “Go home” command, and the automatic homing feature. It also adds several new serial settings: “Alternative device number”, “Enable 14-bit device number”, “Enable 7-bit responses”, “Enable CRC for responses”. Serial commands starting with 0xFF or 0xFE bytes, which previously would cause a serial format error, are now ignored.

Upgrade instructions

You can determine your controller’s firmware version by running the Tic Control Center software, connecting to the controller, and looking in the “Device info” box. If you do not have the latest firmware, you can upgrade the firmware by following these steps:

1. To use the new features added in firmware version 1.06, you will need to upgrade your Tic software to version 1.7.0 or later. You can download the latest version of the Tic software for your operating system from **Section 3**. (The old Tic software will work with the new firmware but will not enable access to the new features.)
2. Save the settings stored on your controller using the “Save settings file…” option in the File menu. All of your settings will be reset to their default values during the firmware upgrade.
3. Download the latest version of the firmware here: **Firmware version 1.06 for the Tic Stepper Motor Controllers [https://www.pololu.com/file/0J1635/tic-v1.06.fmi]** (287k fmi).
4. Run the Tic Control Center application and connect to the controller.
5. In the Device menu, select “Upgrade firmware…”. You will see a message asking you if you are sure you want to proceed: click OK. The Tic will now disconnect itself from your computer, go into bootloader mode, and reappear as a new device.
6. Once the Tic is recognized by the computer, the green LED should be blinking in a double heart-beat pattern.
7. Go to the window titled “Upgrade Firmware” that the Tic Control Center opened. Click the “Browse…” button and select the firmware file you downloaded.
8. If it is not already selected, select the device you want to upgrade from the “Device” dropdown box.
9. Click the “Program” button. You will see a message warning you that your device’s firmware and settings are about to be erased and asking you if you are sure you want to proceed: click OK.
10. It will take a few seconds to erase the Tic’s existing firmware and load the new firmware.
11. Once the upgrade is complete, the Upgrade Firmware window will close, the Tic will disconnect from your computer once again, and it will reappear as it was before. If there is

only one Tic plugged into your computer, the software will connect to it. Check the firmware version number and make sure that it now indicates the latest version of the firmware.

12. If you saved your settings, you can restore them now by using the "Open settings file…" option in the "File" menu and clicking "Apply settings".

If you run into problems during a firmware upgrade, please **contact us** **[https://www.pololu.com/contact]** for assistance.

5.8. Logic power output (5V)

The Tic's **5V (out)** pin provides access to the board's 5V logic supply, which comes from either the USB 5V bus voltage or a 5V regulator powered by VIN, depending on which power source is connected. If power is supplied via VIN and USB at the same time, the Tic uses VIN.

The 5V regulator on the **Tic T500** and **Tic T825** is a low-dropout (LDO) linear regulator. The amount of heat generated by a linear regulator is proportional to the voltage drop from the output to the input multiplied by the output current (which equals the input current). The other main heat-generating component of the Tic is its stepper motor driver, which generates heat in proportion to the current it is supplying to the stepper motor coils. Therefore, to figure out how much current you can draw from the 5V pin without overheating the regulator, you have to consider two factors: the VIN voltage and the Tic's configured motor current limit. The table below shows how much current you can draw from the Tic T825 5V pin at room temperature for various combinations of input voltage and stepper motor current, not including the current drawn by the Tic itself:

VIN voltage	Motor current limit	Current available on Tic T825 5V pin
12 V	0.5 A	70 mA
12 V	1.0 A	70 mA
12 V	1.5 A	70 mA
24 V	0.5 A	70 mA
24 V	1.0 A	70 mA
24 V	1.5 A	50 mA
36 V	0.5 A	70 mA
36 V	1.0 A	50 mA
36 V	1.5 A	30 mA

Note that the available current will be lower at higher ambient temperatures.

Note: While the Tic T500 can operate down to 4.5 V, power supply voltages under 5.5 V could cause a drop in the logic voltage of the board, potentially down to around 4 V when the power supply is 4.5 V. For more information about what effects this can have, see the note in **Section 4.1**.

The 5V regulator on the **Tic T834** is an efficient switching step-up/step-down regulator. At least 450 mA is available on the Tic T834's 5V pin when VIN is in the valid range.

The 5V regulator on the **Tic T249** is an efficient switching step-down regulator. At least 400 mA is available on the Tic T249's 5V pin when VIN is in the valid range.

5.9. Hardware design files

These files provide further documentation of the hardware design of each Tic controller.

Tic T500:

- **Tic T500 dimension diagram [https://www.pololu.com/file/0J1469/tic-t500-usb-multi-interface-stepper-motor-controller-dimensions.pdf]** (198k pdf)
- **Tic T500 3D model [https://www.pololu.com/file/0J1470/tic-t500-usb-multi-interface-stepper-motor-controller.step]** (11MB step)
- **Tic T500 drill guide [https://www.pololu.com/file/0J1468/tic03a-drill.dxf]** (82k dxf)

Tic T834:

- **Tic T834 dimension diagram [https://www.pololu.com/file/0J1377/tic-t834-usb-multi-interface-stepper-motor-controller-dimensions.pdf]** (215k pdf)
- **Tic T834 3D model [https://www.pololu.com/file/0J1378/tic-t834-usb-multi-interface-stepper-motor-controller.step]** (11MB step)
- **Tic T834 drill guide [https://www.pololu.com/file/0J1376/tic02a-drill.dxf]** (71k dxf)

Tic T825:

- **Tic T825 dimension diagram [https://www.pololu.com/file/0J1323/tic-t825-usb-multi-interface-stepper-motor-controller-dimensions.pdf]** (199k pdf)
- **Tic T825 3D model [https://www.pololu.com/file/0J1324/tic-t825-usb-multi-interface-stepper-motor-controller.step]** (12MB step)
- **Tic T825 drill guide [https://www.pololu.com/file/0J1322/tic01a-drill.dxf]** (76k dxf)

Tic T249:

- **Tic T249 dimension diagram [https://www.pololu.com/file/0J1632/tic-t249-usb-multi-interface-stepper-motor-controller-dimensions.pdf]** (348k pdf)
- **Tic T249 3D model [https://www.pololu.com/file/0J1633/tic-t249-usb-multi-interface-stepper-motor-**

controller.step] (12MB step)

- **Tic T249 drill guide [https://www.pololu.com/file/0J1631/tic05a-drill.dxf]** (77k dxf)

6. Setting reference

This section lists all of the settings that the Tic supports. For each setting, this section contains several pieces of information:

- The **Offset** of each setting is the location where it is stored in the Tic's EEPROM memory. You can use this offset with the "Read setting" and "Set setting" commands.
- The **Type** of each setting specifies how many bits the setting occupies, and says whether it is signed or unsigned (if applicable). All the multi-byte settings use little-endian format, meaning that the least-significant byte comes first.
- The **Data** entry for each setting specifies how the data for that setting is encoded in the Tic's memory. Some of the settings lack this field because they are simply dimensionless integers, so their encoding is straightforward.
- The **Default** entry for each setting is the default value it has on a new Tic or a Tic that has been reset to its defaults.
- The **Range** entry for each setting is the allowed range of values for the setting, if applicable. Trying to use a value outside of this range could result in unexpected behavior.
- The **Settings file** entry for each setting is the name of the setting in a Tic settings file, if applicable. You can save and load Tic settings files from the "File" menu of the Tic Control Center, or by using the Tic Command-line Utility (ticcmd).
- The **Settings file data** entry for each setting is the specification of how that setting is encoded in a settings file. Some of the settings lack this field because the encoding is straightforward.
- The **Tic Control Center** entry for each setting is the location of that setting in the Tic Control Center software, if applicable.

List of settings

- **Control mode**
- **Never sleep**
- **Disable safe start**
- **Ignore ERR line high**
- **Automatically clear driver errors**
- **Soft error response**
- **Soft error position**
- **Serial baud rate / serial baud rate generator**

- **Serial device number**
- **Serial alternative device number**
- **Serial enable alternative device number**
- **Serial 14-bit device number**
- **Command timeout**
- **Serial CRC for commands**
- **Serial CRC for responses**
- **Serial 7-bit responses**
- **Serial response delay**
- **VIN measurement calibration**
- **Enable input averaging**
- **Input hysteresis**
- **Input scaling degree**
- **Invert input direction**
- **Input minimum**
- **Input neutral minimum**
- **Input neutral maximum**
- **Input maximum**
- **Target minimum**
- **Target maximum**
- **Encoder prescaler**
- **Encoder postscaler**
- **Enable unbounded position control**
- **SCL pin configuration**
- **SDA pin configuration**
- **TX pin configuration**
- **RX pin configuration**
- **RC pin configuration**
- **Switch polarity map**
- **Kill switch map**

- **Limit switch forward map**
- **Limit switch reverse map**
- **Current limit**
- **Current limit during error**
- **Step mode**
- **Decay mode**
- **AGC mode**
- **AGC bottom current limit**
- **AGC current boost steps**
- **AGC frequency limit**
- **Max speed**
- **Starting speed**
- **Max acceleration**
- **Max deceleration**
- **Invert motor direction**
- **Enable automatic homing**
- **Automatic homing forward**
- **Homing speed towards**
- **Homing speed away**
- **Not initialized**

Control mode

Offset	0x01
Type	unsigned 8-bit
Data	• **0:** Serial / I²C / USB • **1:** STEP/DIR • **2:** RC position • **3:** RC speed • **4:** Analog position • **5:** Analog speed • **6:** Encoder position • **7:** Encoder speed
Default	Serial / I²C / USB
Settings file	`control_mode`
Settings file data	• `serial` • `step_dir` • `rc_position` • `rc_speed` • `analog_position` • `analog_speed` • `encoder_position` • `encoder_speed`
Tic Control Center	Input and motor settings tab, Control mode

The control mode determines what inputs the Tic will use to control the stepper motor, and whether to use position control or speed control.

Never sleep

Offset	Bit 0 of byte 0x02
Type	boolean
Default	false
Settings file	`never_sleep`
Tic Control Center	Advanced settings tab, Miscellaneous box, Never sleep

By default, if the Tic is powered from a USB bus that is in suspend mode (e.g. the computer is sleeping) and VIN power is not present, it will go to sleep to reduce its current consumption and comply with the USB specification. If you enable the "Never sleep" option, the Tic will never go to sleep.

Disable safe start

Offset	Bit 0 of byte 0x03
Type	boolean
Default	false
Settings file	`disable_safe_start`
Tic Control Center	Advanced settings tab, Miscellaneous box, Disable safe start

This option disables the safe start feature, which is described in **Section 5.4**.

Ignore ERR line high

Offset	Bit 0 of byte 0x04
Type	boolean
Default	false
Settings file	`ignore_err_line_high`
Tic Control Center	Advanced settings tab, Miscellaneous box, Ignore ERR line high

This option disables the "ERR line high" error, which is described in **Section 5.4**.

Automatically clear driver errors

Offset	Bit 0 of byte 0x08
Type	boolean
Default	true
Settings file	`auto_clear_driver_error`
Tic Control Center	Advanced settings tab, Miscellaneous box, Automatically clear driver errors

When enabled, this option causes the Tic to periodically clear latched motor driver errors, as described in **Section 5.4**.

Soft error response

Offset	0x53
Type	unsigned 8-bit
Data	• **0:** De-energize • **1:** Halt and hold • **2:** Decelerate to hold • **3:** Go to position
Default	Decelerate to hole
Settings file	`soft_error_response`
Settings file data	`deenergize` , `halt_and_hold` , `decel_to_hold` , or `go_to_position`
Tic Control Center	Advanced settings tab, Soft error response box

This setting sets the soft error response type, as described in **Section 5.4**.

© 2001–2019 Pololu Corporation

Soft error position

Offset	0x54
Type	signed 32-bit
Data	a position in units of microsteps
Default	0
Range	-2,147,483,648 to 2,147,483,647
Settings file	`soft_error_position`
Tic Control Center	Advanced settings tab, Soft error response box, Go to position

If the “Soft error response” setting is set to “Go to position”, this setting determines what position the Tic will go to when a soft error happens. For more information, see **Section 5.4**.

Serial baud rate / serial baud rate generator

Offset	0x06
Type	unsigned 16-bit
Data	12000000 bps divided by desired serial baud rate
Default	9600 bps
Range	200 bps to 115385 bps
Settings file	`serial_baud_rate`
Settings file data	Desired serial baud rate in bits per second
Tic Control Center	Input and motor settings tab, Serial box, Baud rate

The serial baud rate is the speed that the Tic uses for serial communication on its RX and TX lines. The Tic software presents the baud rate as a whole number in units of bits per second (bps). The value that actually is stored in the Tic’s EEPROM memory is a 16-bit unsigned integer called the “Serial baud rate generator”, and it 12000000 bps divided by the desired baud rate, rounded to the nearest whole number.

Serial device number

Offset	0x07 and 0x69
Type	unsigned 14-bit
Default	14
Range	0 to 16383
Settings file	`serial_device_number`
Tic Control Center	Input and motor settings tab, Serial box, Device number

This number serves as the device number to use for Pololu Protocol serial communication and the 7-bit address to use for I²C communication.

This number is stored in two separate pieces in the settings memory: the lower 7 bits of the device number are stored in the lower 7 bits of the settings byte at offset 0x07. The upper 7 bits of the device number are stored in the lower 7 bits of the settings byte at offset 0x69.

This setting was originally a 7-bit setting stored at offset 0x07, but it was expanded to 14 bits in firmware version 1.06.

Serial alternative device number

Offset	0x6A and 0x6B
Type	unsigned 14-bit
Default	0
Range	0 to 16383
Settings file	`serial_alt_device_number`
Tic Control Center	Input and motor settings tab, Serial box, Alternative device number numeric input

This is an alternative device number that the Tic can use for the Pololu Protocol. If the alternative device number is **enabled**, the Tic will respond both to it and to **regular device number**.

This number is stored in two separate pieces in the settings memory: the lower 7 bits of are stored in the lower 7 bits of the settings byte at offset 0x6A. The upper 7 bits are stored in the lower 7 bits of the settings byte at offset 0x6B.

This setting was added in firmware version 1.06.

Serial enable alternative device number

Offset	Bit 7 of byte 0x6A
Type	boolean
Default	0
Settings file	`serial_enable_alt_device_number`
Tic Control Center	Input and motor settings tab, Serial box, Alternative device number checkbox

If this setting is set to true, the Tic will pay attention to Pololu Protocol serial commands that are addressed to the **alternative device number**.

This setting was added in firmware version 1.06.

Command timeout

Offset	0x09
Type	unsigned 16-bit
Data	timeout value in units of milliseconds, or 0 to disable the feature
Default	1000 ms
Range	0 ms to 60000 ms
Settings file	`command_timeout`
Tic Control Center	Input and motor settings tab, Serial box, Enable command timeout

The “Command timeout” setting is the time in milliseconds before the Tic considers it an error if it has not received certain commands. See the description of the “Command timeout” error in **Section 5.4**.

Serial 14-bit device number

Offset	Bit 3 of byte 0x0B
Type	boolean
Default	false
Settings file	`serial_14bit_device_number`
Tic Control Center	Input and motor settings tab, Serial box, Enable 14-bit device number

If this setting is enabled, the Tic will expect two device number bytes in serial commands using the Pololu Protocol, and the maximum device number changes from 127 to 16383. See **Section 9** for details about the Pololu Protocol.

This setting was added in firmware version 1.06.

Serial CRC for commands

Offset	Bit 0 of byte 0x0B
Type	boolean
Default	false
Settings file	`serial_crc_for_commands`
Tic Control Center	Input and motor settings tab, Serial box, Enable CRC for commands

With this option enabled, the Tic requires a 7-bit CRC byte to be appended to every serial command, as described in **Section 9**.

This setting was previously named "Serial CRC enabled" and stored in the settings file as `serial_crc_enabled`.

Serial CRC for responses

Offset	Bit 1 of byte 0x0B
Type	boolean
Default	false
Settings file	`serial_crc_for_responses`
Tic Control Center	Input and motor settings tab, Serial box, Enable CRC for responses

With this option enabled, the Tic appends a 7-bit CRC byte to every response it sends (unless the response is longer than 14 bytes), as described in **Section 9**.

This setting was added in firmware version 1.06.

Serial 7-bit responses

Offset	Bit 2 of byte 0x0B
Type	boolean
Default	false
Settings file	`serial_7bit_responses`
Tic Control Center	Input and motor settings tab, Serial box, Enable CRC for commands

With this option enabled, the Tic encodes its serial responses using bytes between 0 and 0x7F, as described in **Section 9**. This can be useful in setups where the serial response from one Tic will be seen by other Tic devices.

This setting was added in firmware version 1.06.

Serial response delay

Offset	0x5E
Type	unsigned 8-bit
Data	time in units of microseconds
Default	0
Range	0 to 255
Settings file	`serial_response_delay`
Tic Control Center	Input and motor settings, Serial box, Response delay

This setting specifies the minimum amount of time to wait, in microseconds, before processing an I²C byte or sending a serial response. This setting was added in firmware version 1.02.

VIN measurement calibration

Offset	0x14
Type	signed 16-bit
Default	0
Range	−500 to +500
Settings file	`vin_calibration`
Tic Control Center	Advanced settings tab, Miscellaneous box, VIN measurement calibration

This setting adjusts the scaling of the Tic’s VIN measurements. If the Tic’s VIN readings are lower than the actual VIN voltage, you can use a positive value to scale the readings up. If the readings are high, you can use a negative value to scale the readings down. On the Tic T500 and Tic T825, a value of 33 corresponds to a 4% change. On the Tic T834, a value of 9 corresponds to a 4% change. On the Tic T249, a value of 21 corresponds to a 4% change.

Enable input averaging

Offset	Bit 0 of 0x2E
Type	boolean
Default	true
Settings file	`input_averaging_enabled`
Tic Control Center	Input and motor settings tab, Input conditioning box, Enable input averaging

This setting enables RC/analog input averaging as described in **Section 5.2**.

Input hysteresis

Offset	0x2F
Type	unsigned 16-bit
Default	0
Range	0 to 65535
Settings file	`input_hysteresis`
Tic Control Center	Input and motor settings tab, Input conditioning box, Input hysteresis

This setting specifies the amount of hysteresis to apply to RC/analog inputs, as described in **Section 5.2**.

Input scaling degree

Offset	0x20
Type	unsigned 8-bit
Data	• **0:** linear • **1:** quadratic • **2:** cubic
Default	linear
Settings file	`input_scaling_degree`
Settings file data	`linear` , `quadratic` , or `cubic`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Scaling degree

By default, the scaling function used to convert RC and analog inputs into a speed or position is linear. If you set the scaling degree to quadratic or cubic, then the Tic uses a higher-degree polynomial function to scale its RC or analog inputs, which can give you finer control when the input is closer to its neutral position. For more information, see **Section 5.2**.

Invert input direction

Offset	0x21
Type	unsigned 8-bit
Data	• **0:** false • **1:** true
Default	false
Settings file	`input_invert`
Settings file data	`false` or `true`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Invert input direction

By default, lower analog voltages and shorter RC pulse widths correspond to negative positions and speeds. When enabled, this setting flips that correspondence, as described in **Section 5.2**.

Input minimum

Offset	0x22
Type	unsigned 16-bit
Default	0
Range	0 to 4095
Settings file	`input_min`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Input column, Minimum

This is one of the RC/analog input scaling parameters described in **Section 5.2**.

Input neutral minimum

Offset	0x24
Type	unsigned 16-bit
Default	2015
Range	0 to 4095
Settings file	`input_neutral_min`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Input column, Neutral min

This is one of the RC/analog input scaling parameters described in **Section 5.2**.

Input neutral maximum

Offset	0x26
Type	unsigned 16-bit
Default	2080
Range	0 to 4095
Settings file	`input_neutral_max`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Input column, Neutral max

This is one of the RC/analog input scaling parameters described in **Section 5.2**.

Input maximum

Offset	0x28
Type	unsigned 16-bit
Default	4095
Range	0 to 4095
Settings file	`input_max`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Input column, Maximum

This is one of the RC/analog input scaling parameters described in **Section 5.2**.

© 2001–2019 Pololu Corporation

Target minimum

Offset	0x2A
Type	signed 32-bit
Default	−200
Range	−2,147,483,647 to 0
Settings file	`output_min`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Target column, Minimum

This is one of the RC/analog input scaling parameters described in **Section 5.2**.

Target maximum

Offset	0x32
Type	signed 32-bit
Default	200
Range	0 to 2,147,483,647
Settings file	`output_max`
Tic Control Center	Input and motor settings tab, RC and analog scaling box, Target column, Maximum

This is one of the RC/analog input scaling parameters described in **Section 5.2**.

Encoder prescaler

Offset	0x58
Type	unsigned 32-bit
Default	1
Range	1 to 2,147,483,647
Settings file	`encoder_prescaler`
Tic Control Center	Input and motor settings tab, Encoder box, Prescaler

For encoder control modes, this determines the number of encoder counts per unit change of the stepper motor position or speed, as described in **Section 5.3**.

Encoder postscaler

Offset	0x37
Type	unsigned 32-bit
Default	1
Range	1 to 2,147,483,647
Settings file	`encoder_postscaler`
Tic Control Center	Input and motor settings tab, Encoder box, Postscaler

For encoder control modes, this determines the size of a unit change in the stepper motor position or speed, as described in **Section 5.3**.

Enable unbounded position control

Offset	0x5C
Type	unsigned 8-bit
Data	• **0:** false • **1:** true
Default	false
Settings file	`encoder_unlimited`
Settings file data	`false` or `true`
Tic Control Center	Input and motor settings tab, Encoder box, Enable unbounded position control

This setting is described in **Section 5.3**.

SCL pin configuration

Offset	0x3B
Type	8-bit
Data	• **Bits 0–3:** Pin function: ◦ **0:** Default (see below) ◦ **1:** User I/O ◦ **2:** User input ◦ **3:** Potentiometer power (drives high) ◦ **4:** SCL (I²C clock line) ◦ **7:** Kill switch ◦ **8:** Limit switch forward ◦ **9:** Limit switch reverse • **Bits 4–5:** reserved, should be 0 • **Bit 6:** Enable analog readings • **Bit 7:** Enable internal pull-up
Default	0
Settings file	`scl_config`
Settings file data	Space-separated list including a subset of pin options (`pullup` , `analog` , and `active_high`) and one pin function: `default` , `user_io` , `user_input` , `pot_power` , `serial` (SCL), or `kill_switch` .
Tic Control Center	Advanced settings tab, Pin configuration box, SCL

This byte contains most of the configuration of the SCL pin, as described in **Section 5.5**.

If the pin function is set to "Default", then the Tic will ignore **all** of the options set in this byte and just use default settings. The default settings for SCL depend on the control mode: in an analog control mode, SCL is configured as a potentiometer power pin, while in all other control modes, the SCL pin is used as the I²C clock line with its internal pull-up enabled.

SDA pin configuration

Offset	0x3C
Type	8-bit
Data	• **Bits 0–3:** Pin function: ◦ **0:** Default (see below) ◦ **1:** User I/O ◦ **2:** User input ◦ **4:** SDA (I²C data line) ◦ **7:** Kill switch ◦ **8:** Limit switch forward ◦ **9:** Limit switch reverse • **Bits 4–5:** reserved, should be 0 • **Bit 6:** Enable analog readings • **Bit 7:** Enable internal pull-up
Default	0
Settings file	`sda_config`
Settings file data	Space-separated list including a subset of pin options (`pullup` , `analog` , and `active_high`) and one pin function: `default` , `user_io` , `user_input` , `serial` (SDA), or `kill_switch` .
Tic Control Center	Advanced settings tab, Pin configuration box, SDA

This byte contains most of the configuration of the SDA pin, as described in **Section 5.5**.

If the pin function is set to “Default”, then the Tic will ignore **all** of the options set in this byte and just use default settings. The default settings for SDA depend on the control mode: in an analog control mode, SDA is configured as a user input with analog readings enabled, while in all other control modes, SDA is used as the I²C data line with its internal pull-up enabled.

TX pin configuration

Offset	0x3D
Type	8-bit
Data	• **Bits 0–3:** Pin function: ◦ **0:** Default (see below) ◦ **1:** User I/O ◦ **2:** User input ◦ **4:** TX (serial transmit line) ◦ **6:** Encoder input ◦ **7:** Kill switch ◦ **8:** Limit switch forward ◦ **9:** Limit switch reverse • **Bits 4–5:** reserved, should be 0 • **Bit 6:** Enable analog readings • **Bit 7:** Ignored (TX is always pulled up)
Default	0
Settings file	`tx_config`
Settings file data	Space-separated list including a subset of pin options (`analog` and `active_high`) and one pin function: `default` , `user_io` , `user_input` , `serial` (TX), `encoder` , or `kill_switch` .
Tic Control Center	Advanced settings tab, Pin configuration box, TX

This byte contains most of the configuration of the TX pin, as described in **Section 5.5**.

If the pin function is set to “Default”, then the Tic will ignore **all** of the options set in this byte and just use default settings. The default settings for TX depend on the control mode: in an encoder control mode, TX is configured as an encoder input, while in all other control modes, TX is used as the serial transmit line.

RX pin configuration

Offset	0x3E
Type	8-bit
Data	• **Bits 0–3:** Pin function: ◦ **0:** Default (see below) ◦ **1:** User I/O ◦ **2:** User input ◦ **4:** RX (serial receive line) ◦ **6:** Encoder input ◦ **7:** Kill switch ◦ **8:** Limit switch forward ◦ **9:** Limit switch reverse • **Bits 4–5:** reserved, should be 0 • **Bit 6:** Enable analog readings • **Bit 7:** Ignored (RX is always pulled up)
Default	0
Settings file	`rx_config`
Settings file data	Space-separated list including a subset of pin options (`analog` and `active_high`) and one pin function: `default` , `user_io` , `user_input` , `serial` (RX), `encoder` , or `kill_switch` .
Tic Control Center	Advanced settings tab, Pin configuration box, RX

This byte contains most of the configuration of the RX pin, as described in **Section 5.5**.

If the pin function is set to "Default", then the Tic will ignore **all** of the options set in this byte and just use default settings. The default settings for RX depend on the control mode: in an encoder control mode, RX is configured as an encoder input, while in all other control modes, RX is used as the serial receive line.

RC pin configuration

Offset	0x3F
Type	8-bit
Data	• **Bits 0–3:** Pin function: ◦ **0:** Default (see below) ◦ **2:** User input ◦ **5:** RC input ◦ **7:** Kill switch ◦ **8:** Limit switch forward ◦ **9:** Limit switch reverse • **Bits 4–5:** reserved, should be 0 • **Bit 6:** Ignored (cannot do analog readings) • **Bit 7:** Ignored (pin is always pulled down)
Default	0
Settings file	`rc_config`
Settings file data	Space-separated list including a subset of pin options (`active_high` or nothing) and one pin function: `default` , `user_input` , `rc` , or `kill_switch` .
Tic Control Center	Advanced settings tab, Pin configuration box, RC

This byte contains most of the configuration of the RC pin, as described in **Section 5.5**.

If the pin function is set to "Default", then the Tic will use the RC pin as an RC pulse input and measure the durations of pulses received on the line.

Switch polarity map

Offset	0x36
Type	8-bit
Data	• **Bit 0:** polarity of SCL pin if it is used as a switch • **Bit 1:** polarity of SDA pin if it is used as a switch • **Bit 2:** polarity of TX pin if it is used as a switch • **Bit 3:** polarity of RX pin if it is used as a switch • **Bit 4:** polarity of RC pin if it is used as a switch • **Bits 5–7:** reserved, should be 0
Default	0
Settings file	Listed in `scl_config`, `sda_config`, `tx_config`, `rx_config`, `rc_config`
Settings file data	`active_high` or nothing
Tic Control Center	Advanced settings tab, Pin configuration, Active high checkboxes

For each pin configured as a switch, this setting allows you to choose whether that pin is active high or active low. If the pin’s switch polarity bit is 0, it is active low (a low voltage corresponds to an active switch). If the bit is 1, it is active high.

Kill switch map

Offset	0x5D
Type	8-bit
Data	• **Bit 0:** 1 if SCL pin is used as a kill switch • **Bit 1:** 1 if SDA pin is used a kill switch • **Bit 2:** 1 if TX pin is used as a kill switch • **Bit 3:** 1 if RX pin is used as a kill switch • **Bit 4:** 1 if RC pin is used as a kill switch • **Bits 5–7:** reserved, should be 0
Default	0

This byte contains redundant information to make the firmware implementation simpler. Each of the five input pins of the Tic has one bit in this byte. If the bit is 0, it means that the pin is not a kill switch. If the bit is 1, it means that the pin is a kill switch.

Limit switch forward map

Offset	0x5F
Type	8-bit
Data	• **Bit 0:** 1 if SCL pin is used as a forward limit switch • **Bit 1:** 1 if SDA pin is used a forward limit switch • **Bit 2:** 1 if TX pin is used as a forward limit switch • **Bit 3:** 1 if RX pin is used as a forward limit switch • **Bit 4:** 1 if RC pin is used as a forward limit switch • **Bits 5–7:** reserved, should be 0
Default	0

This byte contains redundant information to make the firmware implementation simpler.

Limit switch reverse map

Offset	0x60
Type	8-bit
Data	• **Bit 0:** 1 if SCL pin is used as a reverse limit switch • **Bit 1:** 1 if SDA pin is used a reverse limit switch • **Bit 2:** 1 if TX pin is used as a reverse limit switch • **Bit 3:** 1 if RX pin is used as a reverse limit switch • **Bit 4:** 1 if RC pin is used as a reverse limit switch • **Bits 5–7:** reserved, should be 0
Default	0

This byte contains redundant information to make the firmware implementation simpler.

Current limit

Offset	0x40
Type	unsigned 8-bit
Data	• Tic T500: see current limit code table below • Tic T834 and Tic T825: current limit, in units of 32 mA • Tic T249: current limit, in units of 40 mA
Default	• Tic T500: 174 mA • Tic T834 and Tic T825: 192 mA • Tic T249: 200 mA
Range	• Tic T500: 0 mA to 3093 mA • Tic T834: 0 mA to 3456 mA • Tic T825: 0 mA to 3968 mA • Tic T249: 0 mA to 4480 mA
Settings file	`current_limit`
Settings file data	current limit, in units of mA
Tic Control Center	Input and motor settings, Motor box, Current limit

This setting determines how much current the Tic's motor driver will attempt to generate in the stepper motor coils. For general information about the current limit, see **Section 4.3**.

Note that this setting just sets the default current limit, and it can be temporarily overridden using the "Set current limit" command.

The Tic T500 represents current limits using codes between 0 and 32. The table below shows the correspondence between these current limit codes and the nominal current limit.

Current limit code	Tic T500 current limit
0	0 mA
1	1 mA
2	174 mA
3	343 mA
4	495 mA
5	634 mA
6	762 mA
7	880 mA
8	990 mA
9	1092 mA
10	1189 mA
11	1281 mA
12	1368 mA
13	1452 mA
14	1532 mA
15	1611 mA
16	1687 mA
17	1762 mA
18	1835 mA
19	1909 mA
20	1982 mA
21	2056 mA
22	2131 mA
23	2207 mA

24	2285 mA
25	2366 mA
26	2451 mA
27	2540 mA
28	2634 mA
29	2734 mA
30	2843 mA
31	2962 mA
32	3093 mA

The Tic T834, Tic T825, and Tic T249 represent current limits using numbers between 0 and 124 which are linearly proportional to the current limit. However, due to hardware limitations, not all of these current limits are actually achievable. Disregarding the maximum allowed current limits documented above, the achievable current limits are as follows: 0 through 31, even numbers from 32 to 62, and multiples of four from 64 to 124. If you set a current limit that is not achievable, the Tic will generally accept it and store it in the “Current limit” variable, but it will actually have the same effect as the highest achievable current limit lower than it. For example, a current limit of 33 is the same as a current limit of 32, and a current limit of 66 is the same as a current limit of 64.

Current limit during error

Offset	0x31
Type	unsigned 8-bit
Data	current limit code as specified above for the normal current limit setting, or 0xFF to disable the feature
Default	0
Range	between 0 mA and the normal current limit setting, or disabled
Settings file	`current_limit_during_error`
Settings file data	current limit, in units of mA, or -1 to disable the feature
Tic Control Center	Advanced settings tab, Soft error response, Use different current limit during soft error

If enabled, this setting causes the Tic to use a different stepper motor current limit instead of the normal current limit if an error is happening and the driver is enabled. See **Section 5.4**.

Step mode

Offset	0x41
Type	unsigned 8-bit
Data	• **0:** Full-step • **1:** 1/2 step • **2:** 1/4 step • **3:** 1/8 step • **4:** 1/16 step (Tic T834, Tic T825, and Tic T249 only) • **5:** 1/32 step (Tic T834, Tic T825, and Tic T249 only) • **6:** 1/2 step 100% (Tic T249 only)
Default	0
Settings file	`step_mode`
Settings file data	`1`, `2`, `2_100p`, `4`, `8`, `16`, `32`, `full`, or `half`
Tic Control Center	Input and motor settings tab, Motor box, Step mode

The step mode setting determines how many microsteps add up to one full step. More information about the step modes can be found in **Section 4.3**.

Note that this setting just sets the default step mode, and it can be temporarily overridden using the "Set step mode" command.

Decay mode

Offset	0x42
Type	unsigned 8-bit
Data	• Tic T500: ◦ **0**: Automatic • Tic T834: ◦ **0**: Mixed 50% ◦ **1:** Slow ◦ **2:** Fast ◦ **3:** Mixed 25% ◦ **4:** Mixed 75% • Tic T825: ◦ **0:** Mixed ◦ **1:** Slow ◦ **2:** Fast • Tic T249: ◦ **0**: Mixed
Default	0
Settings file	`decay_mode`
Settings file data	`mixed`, `slow`, `fast`, `mixed25`, `mixed50`, or `mixed75`
Tic Control Center	Input and motor settings tab, Motor box, Decay mode

More information about the Tic T500 automatic decay mode selection can be found in the **MP6500 datasheet [https://www.pololu.com/file/0J1447/MP6500_r1.0.pdf]** (1MB pdf) and **Section 4.3**.

More information about the Tic T834 decay modes can be found in the **DRV8834 datasheet [https://www.pololu.com/file/0J617/drv8834.pdf]** (2MB pdf) and **Section 4.3**.

More information about the Tic T825 decay modes can be found in the **DRV8825 datasheet [https://www.pololu.com/file/0J590/drv8825.pdf]** (1MB pdf) and **Section 4.3**.

More information about the Tic T249's Advanced Dynamic Mixed Decay (ADMD) can be found in the **TB67S249FTG datasheet**.

Note that this setting just sets the default decay mode, and it can be temporarily overridden using the "Set decay mode" command.

© 2001–2019 Pololu Corporation

AGC mode

Offset	0x6C
Type	unsigned 8-bit
Data	• **0:** Off • **1:** On • **2:** Active off
Default	Off
Settings file	`agc_mode`
Settings file data	`off` , `on` , or `active_off`
Tic Control Center	Input and motor settings tab, Motor box, AGC mode

This setting only applies to the Tic T249. It controls whether the Active Gain Control (AGC) feature is enabled or not. The three options are:

- **Off:** AGC is disabled.
- **On:** AGC is enabled.
- **Active off:** AGC is disabled, but if you are using full-step mode (i.e. not using microstepping), the driver inserts a period of time for each coil where the coil’s current is zero, just like it would do if AGC were enabled.

More information about these AGC modes can be found in the **TB67S249FTG application note [https://www.pololu.com/file/0J1521/TB67S279FTG_application_note_en_20180307.pdf]** (1MB pdf).

Note that this setting just sets the default AGC mode, and it can be temporarily overridden using the “Set AGC option” command.

AGC bottom current limit

Offset	0x6D
Type	unsigned 8-bit
Data	• **0:** 45% • **1:** 50% • **2:** 55% • **3:** 60% • **4:** 65% • **5:** 70% • **6:** 75% • **7:** 80%
Default	80%
Settings file	`agc_bottom_current_limit`
Settings file data	`45` , `50` , `55` , `60` , `65` , `70` , `75` , or `80`
Tic Control Center	Input and motor settings tab, Motor box, AGC bottom current limit

This setting only applies to the Tic T249. It controls how much the Active Gain Control (AGC) can reduce the coil current when a light load on the motor is detected. With the default value of 80%, the AGC will only reduce the current limit down to 80% of the configured current limit. You can find more information about this setting in the **TB67S249FTG datasheet**.

Note that this setting just sets the default bottom current limit, and it can be temporarily overridden using the “Set AGC option” command.

AGC current boost steps

Offset	0x6E
Type	unsigned 8-bit
Data	• **0:** 5 steps • **1:** 7 steps • **2:** 9 steps • **3:** 11 steps
Default	5 steps
Settings file	`agc_current_boost_steps`
Settings file data	`5` , `7` , `9` , or `11`
Tic Control Center	Input and motor settings tab, Motor box, AGC current boost steps

This setting only applies to the Tic T249. It controls how long the Active Gain Control (AGC) takes to increase the current in the coils when a heavy load on the motor is detected. With the default value of 5 steps, the AGC will increase the current in 5 steps, which is the quickest option available. You can find more information about this setting in the **TB67S249FTG datasheet**.

Note that this setting just sets the default number of steps, and it can be temporarily overridden using the "Set AGC option" command.

AGC frequency limit

Offset	0x6F
Type	unsigned 8-bit
Data	• **0:** Off • **1:** 225 Hz • **2:** 450 Hz • **3:** 675 Hz
Default	Off
Settings file	`agc_frequency_limit`
Settings file data	`off` , `225` , `450` , or `675`

This setting only applies to the Tic T249. It controls the frequency limit feature provided by the TB67S249FTG driver, which can optionally prevent the the Active Gain Control (AGC) from lowering the coil current if the stepping frequency is too low. The default value of "Off" disables this feature, so there is no lower limit on the stepping frequency needed for the AGC to activate. The other values specify a lower limit on the frequency of *full* steps.

For example, if this setting is 225 Hz, then the AGC will not activate unless the rate of *full* steps is at least 225 Hz. If you are using 1/4 microstepping, this corresponds to a microstep frequency of 900 Hz, which corresponds to a speed of approximately 9,000,000.

You can find more information about this setting in the **TB67S249FTG datasheet**.

Note that this setting just sets the default AGC frequency limit, and it can be temporarily overridden using the "Set AGC option" command.

Max speed

Offset	0x47
Type	unsigned 32-bit
Data	speed in units of pulses per 10,000 seconds
Default	2,000,000 (200 pulses/s)
Range	0 to 500,000,000 (50,000 pulses/s)
Settings file	`max_speed`
Tic Control Center	Input and motor settings, Motor tab, Max speed

The "Max speed" setting is the upper limit on how fast the Tic will try to drive the stepper motor. For more information, see **Section 5.1**.

Note that this setting just sets the default value for the "Max speed" motion parameter, and it can be temporarily overridden using the "Set max speed" command.

Starting speed

Offset	0x43
Type	unsigned 32-bit
Data	speed in units of pulses per 10,000 seconds
Default	0
Range	0 to 500,000,000 (50,000 pulses/s)
Settings file	`starting_speed`
Tic Control Center	Input and motor settings, Motor tab, Starting speed

The “Starting speed” is the maximum speed at which instant acceleration and deceleration are allowed. For more information, see **Section 5.1**.

Note that this setting just sets the default value for the “Starting speed” motion parameter, and it can be temporarily overridden using the “Set starting speed” command.

Max acceleration

Offset	0x4F
Type	unsigned 32-bit
Data	acceleration in units of pulses per second per 100 seconds
Default	40,000
Range	100 to 2,147,483,647
Settings file	`max_accel`
Tic Control Center	Input and motor settings tab, Motor box, Max acceleration

The “Max acceleration” setting specifies how rapidly the speed is allowed to increase. For more information, see **Section 5.1**.

Note that this setting just sets the default value for the “Max acceleration” motion parameter, and it can be temporarily overridden using the “Set max acceleration” command.

Max deceleration

Offset	0x4B
Type	unsigned 32-bit
Data	deceleration in units of pulses per second per 100 seconds, or 0 to make it the same as the max acceleration
Default	0
Range	100 to 2,147,483,647
Settings file	`max_decel`
Tic Control Center	Input and motor settings tab, Motor box, Max deceleration

The "Max deceleration" setting specifies how rapidly the speed is allowed to decrease. For more information, see **Section 5.1**.

Note that this setting just sets the default value for the "Max deceleration" motion parameter, and it can be temporarily overridden using the "Set max deceleration" command. Also, even if this setting is 0 (which corresponds to checking the "Use max acceleration limit for deceleration" box in the Tic Control Center), the "Set max acceleration" command will only change the max acceleration without changing the max deceleration.

Invert motor direction

Offset	Bit 0 of 0x1B
Type	boolean
Default	false
Settings file	`invert_motor_direction`
Tic Control Center	Input and motor settings tab, Motor box, Invert motor direction

By default, an increasing position or a positive speed corresponds to taking steps forward through the motor driver's current indexer table, so the amount of current flowing from B1 to B2 lags behind the amount of current flowing from A1 to A2. This setting flips that correspondence if it is enabled, making the motor turn in the opposite direction. See **Section 5.1**.

Enable automatic homing

Offset	Bit 1 of byte 0x02
Type	boolean
Default	false
Settings file	`auto_homing`
Tic Control Center	Advanced settings tab, Homing box, Enable automatic homing

If true, the Tic will start the homing procedure automatically whenever the “Position uncertain” flag is set, the “Operation state” is “Normal”, and the “Input state” variable (which is set either by the control mode or by serial commands) is “Target position”. This setting is mainly intended for use with “RC Position”, “Analog position”, and “Encoder position” control modes. See **Section 5.6** for more information about homing.

Automatic homing forward

Offset	Bit 2 of byte 0x03
Type	boolean
Default	false
Settings file	`auto_homing_forward`
Tic Control Center	Advanced settings tab, Homing box, Automatic homing direction

If true, the Tic’s automatic homing feature will perform homing in the forward direction. If false, the automatic homing feature will perform homing in the reverse direction.

Homing speed towards

Offset	0x61
Type	unsigned 32-bit
Data	speed in units of pulses per 10,000 seconds
Default	1000000 (100 pulses/s)
Range	0 to 500,000,000 (50,000 pulses/s)
Settings file	`homing_speed_towards`
Tic Control Center	Advanced settings tab, Homing box, Homing speed towards

This is the speed that the Tic uses during the homing procedure when it is travelling towards the limit switch. See **Section 5.6**.

Homing speed away

Offset	0x65
Type	unsigned 32-bit
Data	speed in units of pulses per 10,000 seconds
Default	1000000 (100 pulses/s)
Range	0 to 500,000,000 (50,000 pulses/s)
Settings file	`homing_speed_towards`
Tic Control Center	Advanced settings tab, Homing box, Homing speed away

This is the speed that the Tic uses briefly during the homing procedure when it is travelling away from the limit switch to deactivate it. See **Section 5.6**.

Not initialized

Offset	0x00
Type	unsigned 8-bit
Data	• **0:** false • **non-zero:** true
Default	false

This special setting keeps track of whether the rest of the settings have been initialized or not. Normally it is zero, which means false. If you set it to a non-zero value, then the Tic will reset all of the settings to their default values the next time the Tic is reset or reinitialized. This is how the "Restore default settings" command in the Tic Control Center and the `--restore-defaults` option in the command-line utility are implemented.

7. Variable reference

The Tic maintains a set of variables that contain real-time information about its inputs, outputs, and state, and these variables, in conjunction with the user settings, determine the behavior of the controller. Some of these variables are displayed under the Status tab of the Tic Control Center. Some of the variables can be shown by running the Tic Command-line Utility (ticcmd) with the `-s` or `--status` option, and all the variables are shown if you additionally use the `--full` option (e.g. `ticcmd --status --full`). All variables can also be read via the TTL serial, I²C, and USB interfaces (see the "**get variable [https://www.pololu.com/docs/0J71/8#cmd-get-variable]**" command in **Section 8**) for use by custom control programs, and our Tic library for Arduino includes functions that make it easy to read each variable.

General status variables

<table>
<tr><th>Offset</th><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>0x00</td><td>Operation state</td><td>unsigned 8-bit</td><td>The overall state of the Tic. (See Section 5.4 for descriptions of these states.)
0: Reset
2: De-energized
4: Soft error
6: Waiting for ERR line
8: Starting up
10: Normal</td></tr>
<tr><td>0x01</td><td>Misc flags 1</td><td>unsigned 8-bit</td><td>The set bits of this variable provide additional information about the Tic’s status.
Bit 0: Energized – The Tic’s motor outputs are enabled and if a stepper motor is properly connected, its coils are energized (i.e. electrical current is flowing).
Bit 1: Position uncertain – The Tic has not received external confirmation that the value of its “current position” variable is correct (see Section 5.4).
Bit 2: Forward limit active – One of the forward limit switches is active.
Bit 3: Reverse limit active – One of the reverse limit switches is active.
Bit 4: Homing active – The Tic’s homing procedure is running.
Bits 5–7: reserved</td></tr>
<tr><td>0x02</td><td>Error status</td><td>unsigned 16-bit</td><td>The set bits of this variable indicate the errors that are currently stopping the motor. The motor can only be controlled normally when this variable has a value of 0. (See Section 5.4 for error descriptions.)
Bit 0: Intentionally de-energized
Bit 1: Motor driver error
Bit 2: Low VIN
Bit 3: Kill switch active
Bit 4: Required input invalid
Bit 5: Serial error
Bit 6: Command timeout
Bit 7: Safe start violation</td></tr>
</table>

			• **Bit 8:** ERR line high • **Bits 9–15:** *reserved*
0x04	Errors occurred	unsigned 32-bit	The set bits of this variable indicate the errors that have occurred since this variable was last cleared with the "**get variable and clear errors occurred**" command. • **Bits 0–15:** These bits correspond to the same errors as those of the "error status" variable documented above. • **Bit 16:** Serial framing • **Bit 17:** Serial RX overrun • **Bit 18:** Serial format • **Bit 19:** Serial CRC • **Bit 20:** Encoder skip • **Bits 21–31:** *reserved*

Step planning variables

See **Section 5.1** for a more detailed explanation of the motion parameter variables.

Offset	Name	Type	Description	Units
0x09	Planning mode	unsigned 8-bit	The kind of step planning algorithm the controller is currently using. • **0:** Off (no target; not sending steps) • **1:** Target position • **2:** Target velocity	
0x0A	Target position	signed 32-bit	Motor target position (−2,147,483,648 to +2,147,483,647 = −0x8000 0000 to +0x7FFF FFFF). This value is only meaningful if the “planning mode” variable indicates “target position”.	microsteps
0x0E	Target velocity	signed 32-bit	Motor target velocity (−500,000,000 to +500,000,000). This value is only meaningful if the “planning mode” variable indicates “target velocity”.	microsteps per 10,000 s
0x12	Starting speed	unsigned 32-bit	Maximum speed at which instant acceleration and deceleration are allowed (0 to 500,000,000).	microsteps per 10,000 s
0x16	Max speed	unsigned 32-bit	Maximum allowed motor speed (0 to 500,000,000).	microsteps per 10,000 s
0x1A	Max deceleration	unsigned 32-bit	Maximum allowed motor deceleration (100 to 2,147,483,647 = 0x64 to 0x7FFF FFFF).	microsteps per 100 s^2
0x1E	Max acceleration	unsigned 32-bit	Maximum allowed motor acceleration (100 to 2,147,483,647 = 0x64 to 0x7FFF FFFF).	microsteps per 100 s^2
0x22	Current position	signed 32-bit	Current position of the motor (−2,147,483,648 to +2,147,483,647 = −0x8000 0000 to +0x7FFF FFFF). Note that this just tracks steps that the Tic has commanded the stepper driver to take; it could be different from the actual position of the motor for various reasons.	microsteps
0x26	Current velocity	signed 32-bit	Current velocity of the motor (−500,000,000 to +500,000,000). Note that this is just the step rate and direction the Tic is sending to the driver, and it might not correspond to the actual velocity of the motor for various reasons.	microsteps per 10,000 s

0x2A	Acting target position	signed 32-bit	This is a variable used in the Tic's target position step planning algorithm. It is accessible mainly for getting insight into the algorithm or for troubleshooting.This value could be invalid while the motor is stopped.	microsteps
0x2E	Time since last step	unsigned 32-bit	This is a variable used in the Tic's step planning algorithms. It is accessible mainly for getting insight into the algorithms or for troubleshooting. This value could be invalid while the motor is stopped.	1/3 μs

Other variables

See **Section 6** for details about the step mode and decay mode.

Offset	Name	Type	Description	Units
0x32	Device reset	unsigned 8-bit	The cause of the Tic's last full microcontroller reset. • **0:** Power up • **1:** Brown-out reset • **2:** Reset line ($\overline{RST}$) pulled low by external source • **4:** Watchdog timer reset (should never happen; this could indicate a firmware bug) • **8:** Software reset (by firmware upgrade process) • **16:** Stack overflow (should never happen; this could indicate a firmware bug) • **32:** Stack underflow (should never happen; this could indicate a firmware bug) A "**reset**" command does *not* affect this variable.	
0x33	VIN voltage	unsigned 16-bit	Measured voltage on the VIN pin.	mV
0x35	Up time	unsigned 32-bit	Time since the Tic's microcontroller last experienced a full reset or was powered up. A "**reset**" command does *not* affect this variable.	ms
0x39	Encoder position	signed 32-bit	Raw encoder count measured from the quadrature encoder inputs (TX and RX).	ticks
0x3D	RC pulse width	unsigned 16-bit	Reading from the RC pulse input. 0xFFFF means the reading is not available or invalid.	1/12 μs
0x3F	Analog reading SCL	unsigned 16-bit	Analog reading from the SCL pin, if analog readings are enabled for it. 0xFFFF means the reading is not available.	0 = 0 V, 0xFFFE ≈ voltage on 5V pin
0x41	Analog reading	unsigned 16-bit	Analog reading from the SDA pin, if analog readings are enabled for it. 0xFFFF means	0 = 0 V, 0xFFFE ≈ voltage on

	SDA		the reading is not available.	5V pin
0x43	Analog reading TX	unsigned 16-bit	Analog reading from the TX pin, if analog readings are enabled for it. 0xFFFF means the reading is not available.	0 = 0 V, 0xFFFE ≈ voltage on 5V pin
0x45	Analog reading RX	unsigned 16-bit	Analog reading from the RX pin, if analog readings are enabled for it. 0xFFFF means the reading is not available.	0 = 0 V, 0xFFFE ≈ voltage on 5V pin
0x47	Digital readings	unsigned 8-bit	Digital readings from the Tic's control pins. A set bit indicates that the pin is high. • **Bit 0:** SCL • **Bit 1:** SDA • **Bit 2:** TX • **Bit 3:** RX • **Bit 4:** RC • **Bits 5–7:** *reserved*	
0x48	Pin states	unsigned 8-bit	States of the Tic's control pins, i.e. what kind of input or output each pin is. • **Bits 0–1:** SCL • **Bits 2–3:** SDA • **Bits 4–5:** TX • **Bits 6–7:** RX Each group of two bits encodes a number that represents one of the following states: • **0:** High impedance • **1:** Pulled up • **2:** Output low • **3:** Output high Note that the reported state might be misleading if the pin is being used as a TTL serial or I²C pin. The state of the RC pin cannot be set.	
0x49	Step mode	unsigned 8-bit	Step mode of the Tic's stepper driver (also known as microstepping mode), which defines how many microsteps correspond to one full step.	

			• **0:** Full step • **1:** 1/2 step • **2:** 1/4 step • **3:** 1/8 step • **4:** 1/16 step (Tic T834 and Tic T825 only) • **5:** 1/32 step (Tic T834 and Tic T825 only)	
0x4A	Current limit	unsigned 8-bit	Stepper motor coil current limit of the Tic's stepper driver (0 to 124).	See **Section 6**

0x4B	Decay mode	unsigned 8-bit	Decay mode of the Tic's stepper driver.
			• Tic T500:
			  ◦ **0:** Automatic
			• Tic T834:
			  ◦ **0:** Mixed 50%
			  ◦ **1:** Slow
			  ◦ **2:** Fast
			  ◦ **3:** Mixed 25%
			  ◦ **4:** Mixed 75%
			• Tic T825:
			  ◦ **0:** Mixed
			  ◦ **1:** Slow
			  ◦ **2:** Fast
			• Tic T249:
			  ◦ **0:** Mixed	

0x4C	Input state	unsigned 8-bit	State of the Tic's main input. • **0:** Not ready • **1:** Invalid • **2:** Halt • **3:** Target position • **4:** Target velocity	
0x4D	Input after averaging	unsigned 16-bit	These variables are used in the process that converts raw RC and analog values into a motor position or speed. They are mainly for debugging your input scaling settings in	See **Section 5.2**.
0x4F	Input	unsigned		

	after hysteresis	16-bit	an RC or analog mode. 0xFFFF means the reading is not available.	
0x51	Input after scaling	signed 32-bit	Value of the Tic's main input after scaling has been applied. If the input is valid, this number is the target position or target velocity specified by the input.	**Position:** microsteps **Velocity:** microsteps per 10,000 s
0x55	Last motor driver error	unsigned 8-bit	The cause of the last motor driver error. This variable is valid only for the Tic T249, and will be 0 for other Tic models. • **0:** None • **1:** Over-current • **2:** Over-temperature	
0x56	AGC mode	unsigned 8-bit	See the description of the corresponding setting in **Section 6**.	
0x57	AGC bottom current limit	unsigned 8-bit	See the description of the corresponding setting in **Section 6**.	
0x58	AGC current boost steps	unsigned 8-bit	See the description of the corresponding setting in **Section 6**.	
0x59	AGC frequency limit	unsigned 8-bit	See the description of the corresponding setting in **Section 6**.	

8. Command reference

Overview

All of the Tic's serial commands are available on its TTL serial, I²C, and USB interfaces, with the exception of a few commands that are unique to USB. Each command uses one of these four formats:

1. **Quick** command: no data
2. **7-bit write** command: writes a 7-bit value to the Tic
3. **32-bit write** command: writes a 32-bit value to the Tic
4. **Block read** command: reads a block of data from the Tic; the block starts from the specified offset and can have a variable length

Descriptions of how these command formats are encoded for each interface can be found in the following sections (**Section 9** for TTL serial, **Section 10** for I²C, and **Section 11** for USB). (Note that the USB-only "set setting" command is a special case that uses its own unique format.)

List of commands

- **Set target position**
- **Set target velocity**
- **Halt and set position**
- **Halt and hold**
- **Go home**
- **Reset command timeout**
- **De-energize**
- **Energize**
- **Exit safe start**
- **Enter safe start**
- **Reset**
- **Clear driver error**
- **Set max speed**
- **Set starting speed**
- **Set max acceleration**
- **Set max deceleration**
- **Set step mode**

- **Set current limit**
- **Set decay mode**
- **Set AGC option**
- **Get variable**
- **Get variable and clear errors occurred**
- **Get setting**
- **Set setting** (USB only)
- **Reinitialize** (USB only)
- **Start bootloader** (USB only)

Set target position

Command	0xE0
Format	32-bit write
Data	target position, signed 32-bit • **Range:** −2,147,483,648 to +2,147,483,647 = −0x8000 0000 to +0x7FFF FFFF • **Units:** microsteps
ticcmd	`ticcmd -p NUM` `ticcmd --position NUM`
Arduino library	`tic.setTargetPosition(int32_t position)`

This command sets the target position of the Tic, in microsteps.

If the control mode is set to Serial / I²C / USB, the Tic will start moving the motor to reach the target position. If the control mode is something other than Serial / I²C / USB, this command will be silently ignored.

In the Tic Control Center, the controls in the "Set target" box can be used to set a target position. These controls are on the "Status" tab and are only available when the control mode is Serial / I²C / USB.

Set target velocity

Command	0xE3
Format	32-bit write
Data	target velocity, signed 32-bit • **Range:** −500,000,000 to +500,000,000 • **Units:** microsteps per 10,000 s
ticcmd	`ticcmd -y NUM` `ticcmd --velocity NUM`
Arduino lib	`tic.setTargetVelocity(int32_t velocity)`

This command sets the target velocity of the Tic, in microsteps per 10,000 seconds.

If the control mode is set to Serial / I²C / USB, the Tic will start accelerating or decelerating the motor to reach the target velocity. If the control mode is something other than Serial / I²C / USB, this command will be silently ignored.

In the Tic Control Center, the controls in the "Set target" box can be used to set a target velocity. These controls are on the "Status" tab and are only available when the control mode is Serial / I²C / USB.

Halt and set position

Command	0xEC
Format	32-bit write
Data	current position, signed 32-bit • **Range:** −2,147,483,648 to +2,147,483,647 = −0x8000 0000 to +0x7FFF FFFF • **Units:** microsteps
ticcmd	`ticcmd --halt-and-set-position NUM`
Arduino lib	`tic.haltAndSetPosition(int32_t position)`

This command stops the motor abruptly without respecting the deceleration limit and sets the "Current position" variable, which represents what position the Tic currently thinks the motor is in. Besides stopping the motor and setting the current position, this command also clears the "position uncertain" flag, sets the input state to "halt", and clears the "input after scaling" variable.

If the control mode is something other than Serial / I²C / USB, this command will be silently ignored.

In the Tic Control Center, the "Set current position" button can be used to halt the motor and set the Tic's current position to the specified value. This button is on the "Status" tab and is only available when the control mode is Serial / I²C / USB.

Halt and hold

Command	0x89
Format	Quick
Data	none
ticcmd	`ticcmd --halt-and-hold`
Arduino lib	`tic.haltAndHold()`

This command stops the motor abruptly without respecting the deceleration limit. Besides stopping the motor, this command also sets the "position uncertain" flag (because the abrupt stop might cause steps to be missed), sets the input state to "halt", and clears the "input after scaling" variable.

If the control mode is something other than Serial / I²C / USB, this command will be silently ignored.

In the Tic Control Center, the "Halt motor" button can be used to halt the motor. This button is on the "Status" tab and is only available when the control mode is Serial / I²C / USB.

Go home

Command	0x97
Format	7-bit write
Data	• **0:** Go home in the reverse direction • **1:** Go home in the forward direction
ticcmd	`ticcmd --home rev` or `ticcmd --home fwd`
Arduino lib	`tic.goHomeReverse()` or `tic.goHomeForward()`

Starts the Tic's homing procedure as described in **Section 5.6**.

Reset command timeout

Command	0x8C
Format	Quick
Data	none
ticcmd	`ticcmd --reset-command-timeout`
Arduino lib	`tic.resetCommandTimeout()`

If the command timeout is enabled, this command resets it and prevents the "command timeout" error from happening for some time. See **Section 5.4** for more information about the command timeout.

The Tic Control Center constantly sends this command while it is connected to a Tic.

De-energize

Command	0x86
Format	Quick
Data	none
ticcmd	`ticcmd --deenergize`
Arduino lib	`tic.deenergize()`

This command causes the Tic to de-energize the stepper motor coils by disabling its stepper motor driver. The motor will stop moving and consuming power. This command sets the "position uncertain" flag (because the Tic is no longer in control of the motor's position); the Tic will also set the "intentionally de-energized" error bit, turn on its red LED, and drive its ERR line high.

The "**energize**" command will undo the effect of this command (except it will leave the "position uncertain" flag set) and could make the system start up again.

In the Tic Control Center, the red "De-energize" button in the lower left can be used to de-energize the motor.

© 2001–2019 Pololu Corporation

Energize

Command	0x85
Format	Quick
Data	none
ticcmd	`ticcmd --energize`
Arduino lib	`tic.energize()`

This command is a request for the Tic to energize the stepper motor coils by enabling its stepper motor driver. The Tic will clear the "intentionally de-energized" error bit. If there are no other errors, this allows the system to start up.

In the Tic Control Center, the green "Resume" button in the lower left can be used to attempt to energize the motor.

Exit safe start

Command	0x83
Format	Quick
Data	none
ticcmd	`ticcmd --exit-safe-start`
Arduino lib	`tic.exitSafeStart()`

In Serial / I²C / USB control mode, this command causes the "safe start violation" error to be cleared for 200 ms. If there are no other errors, this allows the system to start up.

In the Tic Control Center, the green "Resume" button in the lower left will additionally exit safe start if the control mode is Serial / I²C / USB.

Enter safe start

Command	0x8F
Format	Quick
Data	none
ticcmd	`ticcmd --enter-safe-start`
Arduino lib	`tic.enterSafeStart()`

If safe start is enabled and the control mode is Serial / I²C / USB, RC speed, analog speed, or encoder speed, this command causes the Tic to stop the motor (using the configured soft error response behavior) and set its "safe start violation" error bit. If safe start is disabled, or if the Tic is not in one of the listed modes, this command will cause a brief interruption in motor control (during which the soft error response behavior will be triggered) but otherwise have no effect.

In Serial mode, an "**exit safe start**" command is required before the Tic will move the motor again. In the speed control modes, the input must be moved to a neutral position before the Tic will exit safe start and move the motor again. See **Section 5.4** for more information about safe start.

Reset

Command	0xB0
Format	Quick
Data	none
ticcmd	`ticcmd --reset`
Arduino lib	`tic.reset()`

This command makes the Tic forget most parts of its current state. Specifically, it does the following:

- Reloads all settings from the Tic's non-volatile memory and discards any temporary changes to the settings previously made with serial commands (this applies to the step mode, current limit, decay mode, max speed, starting speed, max acceleration, and max deceleration settings)
- Abruptly halts the motor
- Resets the motor driver
- Sets the Tic's operation state to "reset"

- Clears the last movement command and the current position
- Clears the encoder position
- Clears the serial and "command timeout" errors and the "errors occurred" bits
- Enters safe start if configured to do so

The Tic's serial and I²C interfaces will be unreliable for a brief period after the Tic receives the Reset command, so we recommend waiting at least 10 ms after sending a Reset command before sending additional I²C or serial commands.

This command does not modify the "intentionally de-energized" error bit. Since this command does not trigger a full microcontroller reset, it does *not* affect the "up time" variable (which indicates the time elapsed since the Tic's last full microcontroller reset) or the "device reset" variable (which indicates the cause of the Tic's last full microcontroller reset).

Clear driver error

Command	0x8A
Format	Quick
Data	none
ticcmd	`ticcmd --clear-driver-error`
Arduino lib	`tic.clearDriverError()`

This command attempts to clear a motor driver error, which is an over-current or over-temperature fault reported by the Tic's motor driver. If the "Automatically clear driver errors" setting is enabled (the default), the Tic will automatically clear a driver error and it is not necessary to send this command. Otherwise, this command must be sent to clear the driver error before the Tic can continue controlling the motor. See **Section 5.4** for more information about driver errors.

Set max speed

Command	0xE6
Format	32-bit write
Data	max speed, unsigned 32-bit • **Range:** 0 to 500,000,000 • **Units:** microsteps per 10,000 s
ticcmd	`ticcmd --max-speed NUM`
Arduino lib	`tic.setMaxSpeed(uint32_t speed)`

This command temporarily sets the Tic's maximum allowed motor speed in units of steps per 10,000 seconds. The provided value will override the corresponding setting from the Tic's non-volatile memory until the next **Reset** (or **Reinitialize**") command or full microcontroller reset.

Set starting speed

Command	0xE5
Format	32-bit write
Data	starting speed, unsigned 32-bit • **Range:** 0 to 500,000,000 • **Units:** microsteps per 10,000 s
ticcmd	`ticcmd --starting-speed NUM`
Arduino lib	`tic.setStartingSpeed(uint32_t speed)`

This command temporarily sets the Tic's starting speed in units of steps per 10,000 seconds. This is the maximum speed at which instant acceleration and deceleration are allowed; see **Section 5.1** for more information. The provided value will override the corresponding setting from the Tic's non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset.

Set max acceleration

Command	0xEA
Format	32-bit write
Data	max accel, unsigned 32-bit • **Range:** 100 to 2,147,483,647 = 0x64 to 0x7FFF FFFF • **Units:** microsteps per 100 s²
ticcmd	`ticcmd --max-accel NUM`
Arduino lib	`tic.setMaxAccel(uint32_t accel)`

This command temporarily sets the Tic's maximum allowed motor acceleration in units of steps per second per 100 seconds. The provided value will override the corresponding setting from the Tic's non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset.

If the provided value is between 0 and 99, it is treated as 100.

This command does not affect the max deceleration, even if the "Use max acceleration limit for deceleration" checkbox in the Tic Control Center is checked. See the **set max deceleration** command.

Set max deceleration

Command	0xE9
Format	32-bit write
Data	max decel, unsigned 32-bit • **Range:** 100 to 2,147,483,647 = 0x64 to 0x7FFF FFFF • **Units:** microsteps per 100 s²
ticcmd	`ticcmd --max-decel NUM`
Arduino lib	`tic.setMaxDecel(uint32_t decel)`

This command temporarily sets the Tic's maximum allowed motor deceleration in units of steps per second per 100 seconds. The provided value will override the corresponding setting from the Tic's non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset.

If the provided value is 0, then the max deceleration will be set equal to the current max acceleration

value. If the provided value is between 1 and 99, it is treated as 100.

Set step mode

Command	0x94
Format	7-bit write
Data	step mode, unsigned 7-bit • **0:** Full step • **1:** 1/2 step • **2:** 1/4 step • **3:** 1/8 step • **4:** 1/16 step (Tic T834 and Tic T825 only) • **5:** 1/32 step (Tic T834 and Tic T825 only) • **6:** 1/2 step 100% (Tic T249 only)
ticcmd	`ticcmd --step-mode MODE` (full, half, 1, 2, 2_100p, 4, 8, 16, 32)
Arduino lib	`tic.setStepMode(TicStepMode mode)` (`TicStepMode::Full` , `TicStepMode::Half` , `TicStepMode::Microstep1` , …, `TicStepMode::Microstep32`)

This command temporarily sets the step mode (also known as microstepping mode) of the driver on the Tic, which defines how many microsteps correspond to one full step. The provided value will override the corresponding setting from the Tic’s non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset.

Set current limit

Command	0x91
Format	7-bit write
Data	current limit, unsigned 7-bit • **Range:** 0 to 124 • **Units:** See **Section 6**
ticcmd	`ticcmd --current NUM` (in **mA**)
Arduino lib	`tic.setCurrentLimit(uint16_t limit)` (in **mA**)

This command temporarily sets the stepper motor coil current limit of the driver on the Tic in units of 32 milliamps. The provided value will override the corresponding setting from the Tic’s non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset.

Note that the Tic Command-line Utility and the Tic Arduino library use units of milliamps instead.

Set decay mode

Command	0x92
Format	7-bit write
Data	decay mode, unsigned 7-bit • Tic T500: ◦ **0:** Automatic • Tic T834: ◦ **0:** Mixed 50% ◦ **1:** Slow ◦ **2:** Fast ◦ **3:** Mixed 25% ◦ **4:** Mixed 75% • Tic T825: ◦ **0:** Mixed ◦ **1:** Slow ◦ **2:** Fast • Tic T249: ◦ **0:** Mixed
ticcmd	`ticcmd --decay MODE` (mixed, slow, fast, mixed25, mixed50, mixed75)
Arduino lib	`tic.setDecayMode(TicDecayMode mode)` (`TicDecayMode::Mixed` , `TicDecayMode::Slow` , `TicDecayMode::Fast` , `TicDecayMode::Mixed25` , `TicDecayMode::Mixed50` , `TicDecayMode::Mixed75`)

This command temporarily sets the decay mode of the driver on the Tic. For more information about the decay mode, see **Section 6** and the driver datasheet. The provided value will override the corresponding setting from the Tic’s non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset. If the command contains an unrecognized decay mode, the Tic will use decay mode 0. Although the decay mode on the Tic T500 and Tic T249 is not configurable, those Tics still accept this command.

Set AGC option

<table>
<tr><th>Command</th><td>0x98</td></tr>
<tr><th>Format</th><td>7-bit write</td></tr>
<tr><th>Data</th><td>Upper 3 bits specify which AGC option to temporarily change:
0: AGC mode
1: AGC bottom current limit
2: AGC current boost steps
3: AGC frequency limit
The lower 4 bits specify the new value of that option. See the documentation of the corresponding settings in Section 6.</td></tr>
<tr><th>ticcmd</th><td><code>ticcmd --agc-mode MODE</code> (on, off, active_off)
<code>ticcmd --agc-bottom-current-limit LIMIT</code> (45, 50, 55, 60, 65, 70, 75, 80)
<code>ticcmd --agc-current-boost-steps STEPS</code> (5, 7, 9, 11)
<code>ticcmd --agc-frequency-limit LIMIT</code> (off, 225, 450, 675)</td></tr>
<tr><th>Arduino lib</th><td><code>tic.setAgcMode(TicAgcMode mode)</code>
<code>tic.setAgcBottomCurrentLimit(TicAgcBottomCurrentLimit limit)</code>
<code>tic.setAgcCurrentBoostSteps(TicAgcCurrentBoostSteps steps)</code>
<code>tic.setAgcFrequencyLimit(TicAgcFrequencyLimit limit)</code>
(see Tic.h for allowed enum values)</td></tr>
</table>

This command is only valid for the Tic T249. It temporarily changes one of the configuration options of the Active Gain Control (AGC). The provided value will override the corresponding setting from the Tic's non-volatile memory until the next **Reset** (or **Reinitialize**) command or full microcontroller reset.

Get variable

Command	0xA1
Format	Block read
Returns	block of data from variables

ticcmd	`ticcmd -s`
	`ticcmd --status`
	(shows info with important variables)

	`ticcmd -s --full`
	`ticcmd --status --full`
	(shows info with all variables)

Arduino lib	(see the `tic.get____()` functions)

This command reads a block of data from the Tic’s variables; the block starts from the specified offset and can have a variable length. See **Section 7** for the offset and type of each variable.

Get variable and clear errors occurred

Command	0xA2
Format	Block read
Returns	block of data from variables
Arduino lib	(see `tic.getErrorsOccurred()`)

This command is identical to the **Get variable** command, except that it also clears the “Errors occurred” variable at the same time. The intended use of this command is to both read and clear the “errors occurred” variable so that whenever you see a bit set in that variable, you know it indicates an error that occurred since the last “Get variable and clear errors occurred” command.

Get setting

Command	0xA8
Format	Block read
Returns	block of data from settings
ticcmd	`ticcmd --get-settings FILE` (reads all settings from device to file)
Arduino lib	`tic.getSetting(uint8_t offset, uint8_t length, uint8_t * buffer)`

This command reads a block of data from the Tic's settings (stored in non-volatile memory); the block starts from the specified offset and can have a variable length. While some of the Tic's settings can be overridden temporarily using other commands documented in this section, the settings that this command accesses are stored in non-volatile memory and can only be changed using the **Set setting** USB command. See **Section 6** for the offset and type of each setting.

Set setting (USB only)

Command	0x13
Format	special (see **Section 11**)
Data	setting data, 8-bit
ticcmd	`ticcmd --settings FILE` (writes all settings from file to device)

This command writes a byte of data to the Tic's settings (stored in non-volatile memory) at the specified offset. It is not available on the TTL serial and I²C interfaces. The Tic Control Center software uses this command when you click the "Apply settings" button.

Be careful not to call this command in a fast loop to avoid wearing out the Tic's EEPROM, which is rated for only 100,000 write cycles.

Reinitialize (USB only)

Command	0x10
Format	Quick
Data	none

This command reloads all settings from the Tic's non-volatile memory and discards any temporary changes to the settings previously made with serial commands (this applies to the step mode, current limit, decay mode, max speed, starting speed, max acceleration, and max deceleration settings). It does not have any of the other effects of the **Reset** command, instead applying the new settings seamlessly if possible. This command is not available on the TTL serial and I²C interfaces.

Start bootloader (USB only)

Command	0xFF
Format	Quick
Data	none

This command causes the Tic to start its bootloader in preparation for receiving a firmware upgrade over USB. It is not available on the TTL serial and I²C interfaces.

9. Serial command encoding

As described in **Section 8**, the Tic's serial commands each use one of four formats: **quick**, **7-bit write**, **32-bit write**, and **block read**. This section explains how these four command formats are encoded as sequences of bytes within TTL serial command packets.

The following examples are shown using the **compact protocol**, which is intended for cases where the Tic is the only device connected to your serial line. Later on, this section also describes the **Pololu protocol**, which can be used to daisy-chain a Tic on a single serial line with other devices (including additional Tics).

For a reference implementation of these protocols, see the TicSerial class in our **Tic Stepper Motor Controller library for Arduino [https://github.com/pololu/tic-arduino]**.

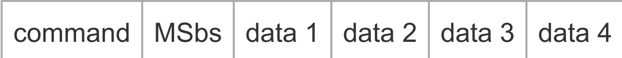
Quick

A quick command sends no data and simply consists of the command byte.

7-bit write

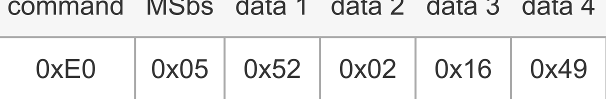
A 7-bit write command has a single byte of data containing a 7-bit value (the most-significant bit of the data byte is always 0) that comes after the command byte.

32-bit write

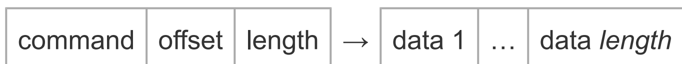


command	MSbs	data 1	data 2	data 3	data 4

A 32-bit write command has five data bytes encoding a 32-bit value. Bits 0 through 3 of the first byte contain the most-significant bits (MSbs) for the four data bytes that follow: bit 0 is the MSb for the least-significant data byte and bit 3 is the MSb for the most-significant data byte. The last four data bytes contain the 32-bit value in little-endian order (starting with the least-significant byte), but with the most-significant bit of each byte cleared.

> **Example: Set target position [https://www.pololu.com/docs/0J71/8#cmd-set-target-position]** to 1,234,567,890
>
> 
>
> | command | MSbs | data 1 | data 2 | data 3 | data 4 |
> |---|---|---|---|---|---|
> | 0xE0 | 0x05 | 0x52 | 0x02 | 0x16 | 0x49 |
>
> Write value: 1,234,567,890 = **0**100 1001 **1**001 0110 **0**000 0010 **1**101 0010
>
> - MSbs: 0000 **0101** = 0x05
> - data 1: 0101 0010 = 0x52
> - data 2: 0000 0010 = 0x02
> - data 3: 0001 0110 = 0x16
> - data 4: 0100 1001 = 0x49

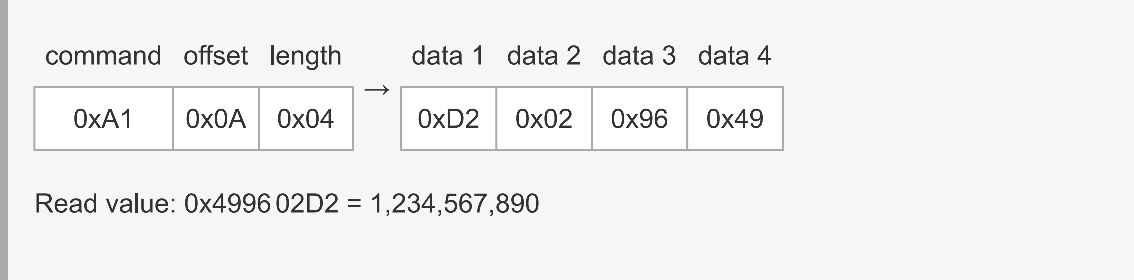
Block read



command	offset	length	→	data 1	…	data *length*

A block read command reads a block of data from the Tic. The offset byte specifies the offset within the data that the response should start at, and the length byte specifies how many bytes of data the response should include. The length must be between 1 and 15. Any multi-byte values contained in the response are in little-endian order (starting with the least-significant byte).

> **Example: Get variable [https://www.pololu.com/docs/0J71/8#cmd-get-variable]** "target position" (32 bits)



command	offset	length	→	data 1	data 2	data 3	data 4
0xA1	0x0A	0x04		0xD2	0x02	0x96	0x49

Read value: 0x4996 02D2 = 1,234,567,890

Serial protocols

Like many other Pololu products, the Tic supports two different serial command protocols.

The **compact protocol** is the simpler of the two protocols; it is the protocol you should use if your Tic is the only device connected to your serial line. The compact protocol command packet is simply a command byte followed by any data bytes that the command requires. All of the examples above use the compact protocol; here is the example for the **Set step mode** **[https://www.pololu.com/docs/0J71/8#cmd-set-step-mode]** command again (selecting 1/8 step mode):

command	data
0x94	0x03

Notice that a command byte always has its most significant bit set (it is in the range 0x80 to 0xFF), while a data byte sent to the Tic always has its most significant bit cleared (it is in the range 0x00 to 0x7F); this is why a 32-bit write command requires five bytes to represent a 32-bit number. Responses from the Tic can contain data bytes of any value from 0x00 to 0xFF.

The **Pololu protocol** can be used in situations where you have multiple devices connected to your serial line. This protocol is compatible with the serial protocol used by our other serial motor and servo controllers. As such, you can daisy-chain a Tic on a single serial line along with our other serial controllers (including additional Tics) and, using this protocol, send commands specifically to the desired Tic without confusing the other devices on the line.

To use the Pololu protocol, you must transmit **0xAA** (170 in decimal) as the first (command) byte, followed by a device number data byte. The default device number for the Tic is **0x0E** (**14** in decimal), but this is a setting you can change. (The device number is also used as the Tic’s I²C slave address.) Any controller on the line whose device number matches the specified device number accepts the command that follows; all other Pololu devices ignore the command. The remaining bytes in the command packet are the same as the compact protocol command packet you would send, with one key difference: the compact protocol command byte is now a data byte for the command 0xAA and hence **must have its most significant bit cleared**. Therefore, the same “set step mode” command from above, but using the Pololu protocol, looks like this:

	device #	command	data
0xAA	0x0E	0x14	0x03

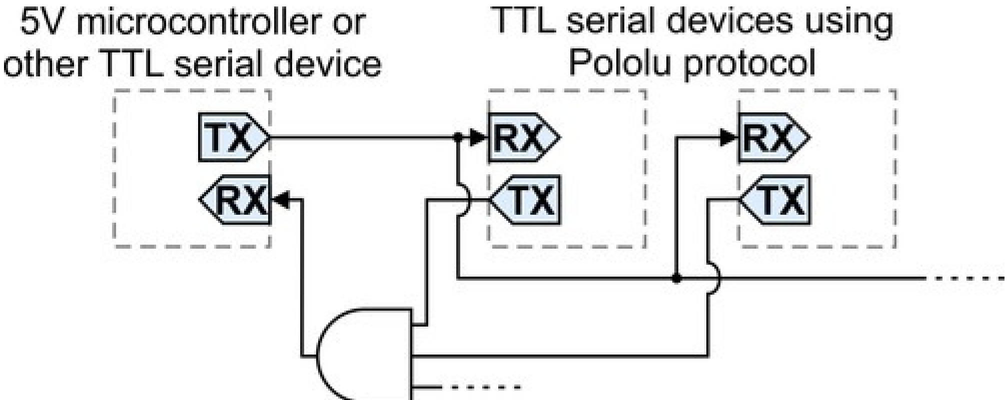
The byte 0x14 is the "Set step mode" command (0x94) with its most significant bit cleared.

The Tic can be configured to respond to an **alternative device number**. If the alternative device number is enabled, the Tic will respond to commands that are addressed it, in addition to commands addressed to the regular device number. This can be useful if you want to assign your Tics to groups and send a single command to all the Tics in the group, while still being able to individually address each Tic. You can enable the alternative device number by checking the checkbox labeled "Alternative device number" in the "Input and motor settings" tab and then entering the number in the corresponding numeric input. This feature was added in firmware version 1.06.

The Tic can be configured to use **14-bit device numbers**. With this option enabled, the device numbers can range from 0 to 16383, instead of the typical range of 0 to 127. After the 0xAA byte, you have to send two device number bytes, both of which are between 0 and 127: the first byte holds the lower 7 bits of the device number, while the second byte holds the upper 7 bits of the device number. You can enable 14-bit device numbers by checking the checkbox labeled "Enable 14-bit device number" in the "Input and motor settings" tab. This feature is useful if you want your system to have more than 128 devices. This feature was added in firmware version 1.06.

The TicSerial class in our Tic Arduino library uses the compact protocol by default if the optional `deviceNumber` argument is omitted, but you can make it use the Pololu protocol instead by providing a device number. See the library documentation for more details.

The diagram below shows how to connect multiple devices that support the Pololu protocol and control them from the serial interface of a single 5V microcontroller or other serial device. The AND gate is only needed if you want to read data back from more than one device.



Daisy-chaining multiple TTL serial devices that support the Pololu protocol for control by a single microcontroller.

Cyclic Redundancy Check (CRC) error detection

For certain applications, verifying the integrity of the data you are sending and receiving can be very important. Because of this, the Tic has optional 7-bit cyclic redundancy checking (CRC). Given a series of bytes, the Tic's CRC algorithm generates a 7-bit number between 0 and 127 called the CRC byte, which is similar to a checksum but more robust as it can detect errors that would not affect a checksum, such as an extra zero byte or bytes out of order.

The Tic has two options for enabling CRC in the "Input and and motor settings" tab of the Tic Control Center.

The **Enable CRC for commands** option makes the Tic expect a CRC byte to be added onto the end of every command packet. The CRC byte is computed by applying the CRC algorithm described below to the other bytes of the command packet, including the 0xAA byte and the device number if you are using the Pololu Protocol. If the CRC byte is incorrect, a CRC error will occur and the command will be ignored.

The **Enable CRC for responses** option makes the Tic append a CRC byte onto the end of any response that it sends. The CRC byte is between 0 and 127, and is computed by applying the CRC algorithm described below to the other bytes of the response. You can check the CRC byte when processing the response from the Tic. The maximum response size supported by the Tic is 15 bytes, so it will not append a CRC byte onto responses that are already 15 bytes long. This behavior might change in future firmware versions. We recommend that you request at most 14 bytes of data when using this option, so that the CRC byte can always be appended. The "Enable CRC for responses" option was added in firmware version 1.06.

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 Tic 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:

```
#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 compact protocol "Set step mode" command selecting 1/8 step mode with a CRC byte appended to it would be:

Compact protocol with CRC:	0x94	0x03	0x10

7-bit responses

In general, the data sent by the Tic in response to serial commands is arbitrary binary data: the bytes can be anything from 0 to 255. The Tic's "Enable 7-bit responses" option causes it to encode those responses in a format that only uses values from 0 to 127. This can be useful if you have wired your Tics in such a way that the responses from one Tic will be seen by another Tic, and you do not want those responses to accidentally be interpreted as commands. This feature was added in firmware version 1.06.

The 7-bit encoding performed by the Tic works as follows. First, if the serial response is longer than 7 bytes, the Tic truncates it to be exactly 7 bytes long, throwing away all the bytes after the first 7. This behavior might change in future firmware versions, so we recommend that you do not request more than 7 bytes from the Tic if you have enabled 7-bit responses. Next, the Tic sets the most-significant bit of each byte in the response to 0. The Tic packs the former values of those most significant bits (MSbs) into a single byte and appends it to the response. The bits of that final byte are in the same

order as the bytes they correspond to: the least significant bit corresponds to the first byte, while the next bit corresponds to the second byte, and so on. The Tic does this conversion before it computes and appends the optional CRC byte.

Below is some C code you can use to undo the 7-bit encoding and restore the MSbs to their respective bytes. The length argument should be length of the entire 7-bit response, including the final byte with the MSbs, but not including the CRC byte (if there is one). After this function runs, the byte holding the MSbs is redundant and can be ignored.

```
void undo7BitEncoding(uint8_t * message, uint8_t length)
{
  uint8_t msbs = message[length - 1];
  for (uint8_t i = 0; i < length - 1; i++)
  {
    if (msbs & 1) { message[i] |= 0x80; }
    msbs >>= 1;
  }
}
```

10. I^2C command encoding

As described in **Section 8**, the Tic’s I^2C commands each use one of four formats: **quick**, **7-bit write**, **32-bit write**, and **block read**. This section explains how these four command formats are encoded as sequences of bytes within I^2C transfers.

For a reference implementation of this protocol, see the TicI2C class in our **Tic Stepper Motor Controller library for Arduino [https://github.com/pololu/tic-arduino]**.

Note that most of these formats are compatible with the SMBus protocols of the same names; an exception is the block read command, although an SMBus-compatible workaround can be found in its description below.

The default slave address for the Tic is **0001110b** (**0x0E** in hex; **14** in decimal), but this is a setting (“device number”) you can change. (This setting also determines the Tic’s device number when using the Pololu protocol over TTL serial.)

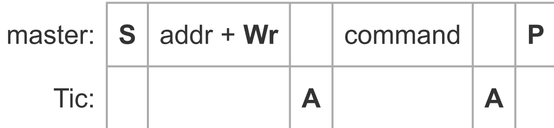
As specified by the I^2C standard, a transfer’s address byte consists of the 7-bit slave address followed by another bit to indicate the transfer direction: 0 for writing to the slave, 1 for reading from the slave. This is denoted by “addr + **Wr**” and “addr + **Rd**” below. With the Tic’s default slave address, the address byte is 0001110**0**b (0x1C) for a write transfer and 0001110**1**b (0x1D) for a read transfer.

These symbols are also used in the format descriptions below:

- **S**: start condition
- **P**: stop condition
- **A**: acknowledge (ACK)
- **N**: not acknowledge (NACK)

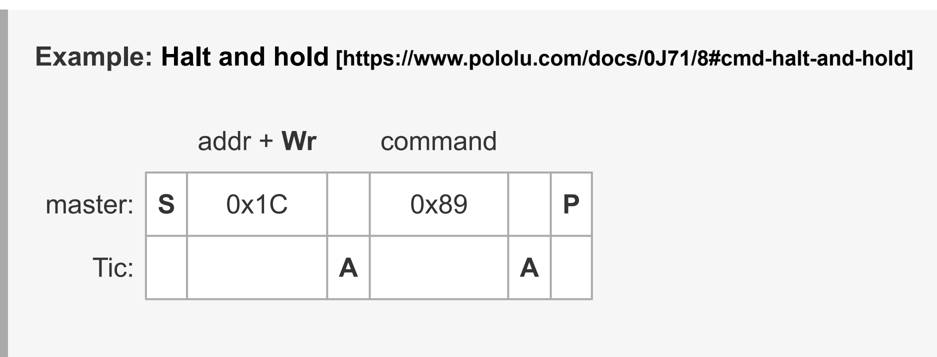
Any stop condition followed by a start condition can optionally be replaced with a repeated start condition.

Quick



master:	**S**	addr + **Wr**		command		**P**
Tic:			**A**		**A**	

A Quick command sends no data and simply consists of the command byte.



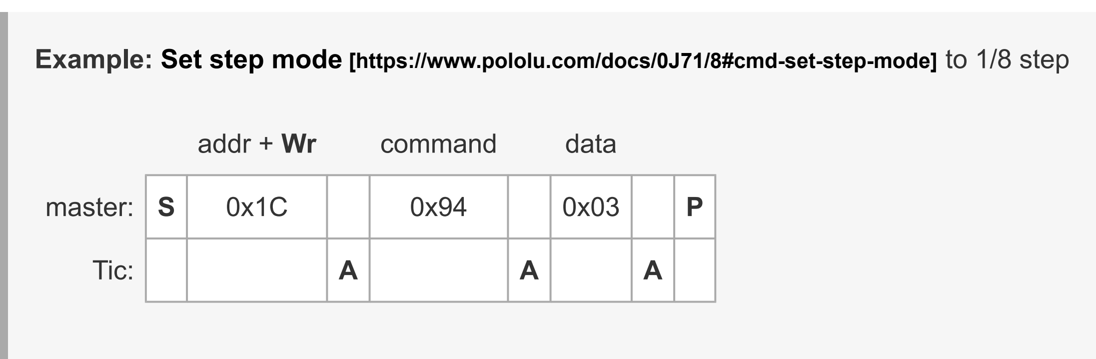
Example: **Halt and hold** **[https://www.pololu.com/docs/0J71/8#cmd-halt-and-hold]**

		addr + **Wr**		command		
master:	**S**	0x1C		0x89		**P**
Tic:			**A**		**A**	

7-bit write

master:	**S**	addr + **Wr**		command		data		**P**
Tic:			**A**		**A**		**A**	

A 7-bit write command has a single byte of data containing a 7-bit value (the most-significant bit of the data byte is always 0).



Example: **Set step mode** **[https://www.pololu.com/docs/0J71/8#cmd-set-step-mode]** to 1/8 step

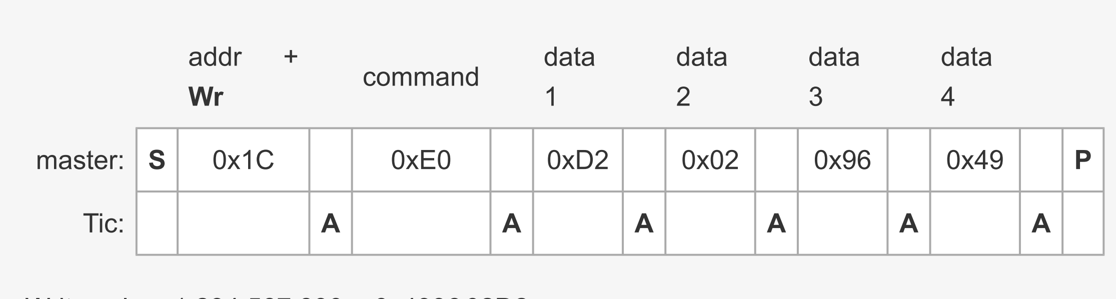
		addr + **Wr**		command		data		
master:	**S**	0x1C		0x94		0x03		**P**
Tic:			**A**		**A**		**A**	

32-bit write

master:	**S**	addr + **Wr**		command		data1		data 2		data 3		data 4		**P**
Tic:			**A**		**A**		**A**		**A**		**A**		**A**	

A 32-bit write command has four data bytes following the command byte. These contain the 32-bit value in little-endian order (starting with the least-significant byte).

Example: **Set target position** **[https://www.pololu.com/docs/0J71/8#cmd-set-target-position]** to 1,234,567,890



		addr + **Wr**		command		data 1		data 2		data 3		data 4		
master:	**S**	0x1C		0xE0		0xD2		0x02		0x96		0x49		**P**
Tic:			**A**		**A**		**A**		**A**		**A**		**A**	

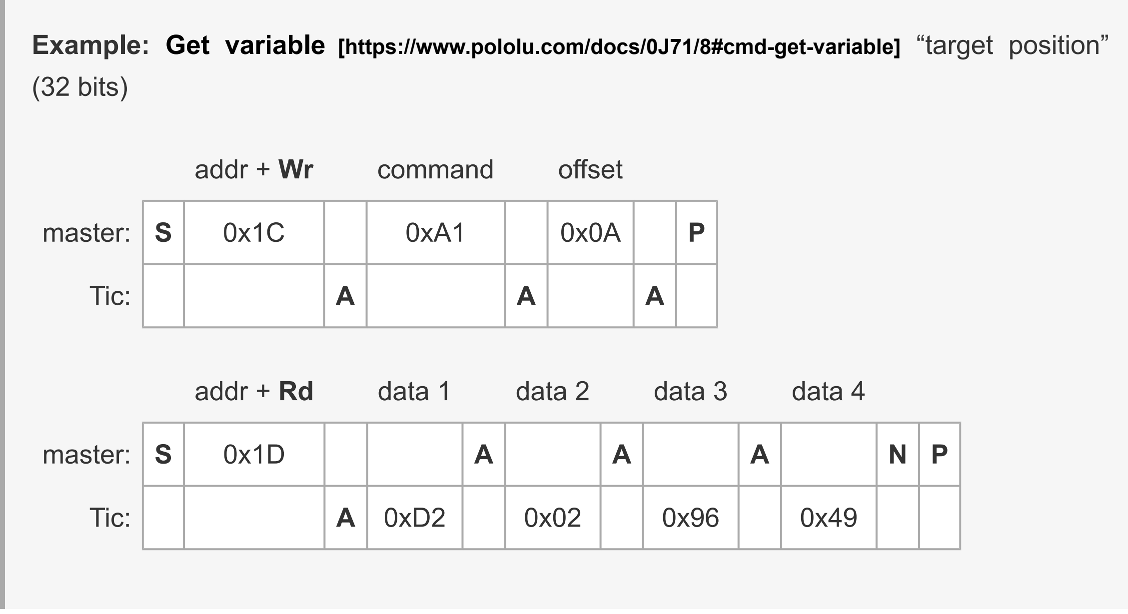
Write value: 1,234,567,890 = 0x4996 02D2

Block read

master:	**S**	addr + **Wr**		command		offset		**P**
Tic:			**A**		**A**		**A**	

master:	**S**	addr + **Rd**			**A**	…		**N**	**P**
Tic:			**A**	data 1		…	data *n*		

A block read command reads a block of data from the Tic. The offset byte specifies the offset within the data that the response should start at. The master can then read up to 15 bytes of response data from the Tic. Any multi-byte values contained in the response are in little-endian order (starting with the least-significant byte).



Example: Get variable **[https://www.pololu.com/docs/0J71/8#cmd-get-variable]** “target position” (32 bits)

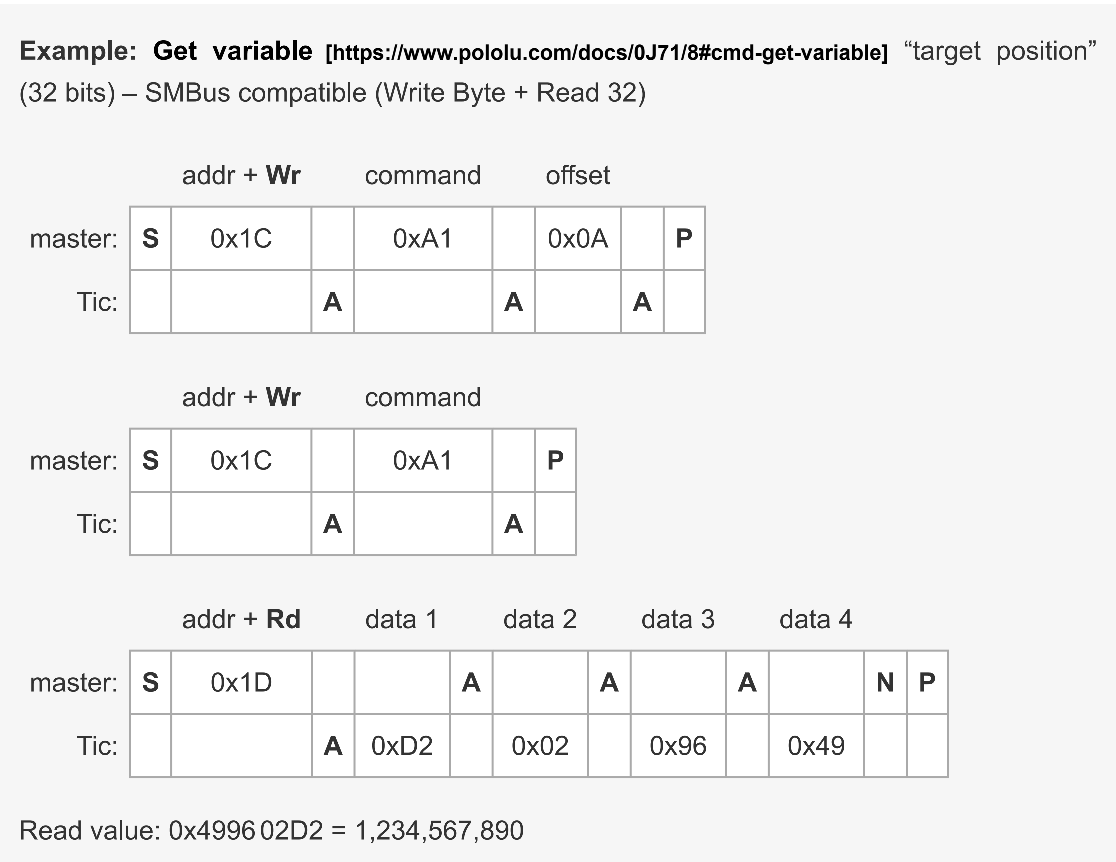
		addr + **Wr**		command		offset		
master:	**S**	0x1C		0xA1		0x0A		**P**
Tic:			**A**		**A**		**A**	

		addr + **Rd**		data 1		data 2		data 3		data 4		
master:	**S**	0x1D			**A**		**A**		**A**		**N**	**P**
Tic:			**A**	0xD2		0x02		0x96		0x49		

 © 2001–2019 Pololu Corporation

> Read value: 0x4996 02D2 = 1,234,567,890

Unlike the other Tic command formats, the block read command is not compatible with the SMBus protocol of the same name. If your master device is limited to using SMBus protocols, you can do a Tic block read by first using an SMBus "Write Byte" transfer to send the Tic command and offset, then using an SMBus read transfer (e.g. Read Byte, Read Word, or Read 32) to send the same Tic command and get the response data:



Example: Get variable **[https://www.pololu.com/docs/0J71/8#cmd-get-variable]** "target position" (32 bits) – SMBus compatible (Write Byte + Read 32)

		addr + **Wr**		command		offset		
master:	**S**	0x1C		0xA1		0x0A		**P**
Tic:			**A**		**A**		**A**	

		addr + **Wr**		command		
master:	**S**	0x1C		0xA1		**P**
Tic:			**A**		**A**	

		addr + **Rd**		data 1		data 2		data 3		data 4		
master:	**S**	0x1D			**A**		**A**		**A**		**N**	**P**
Tic:			**A**	0xD2		0x02		0x96		0x49		

Read value: 0x4996 02D2 = 1,234,567,890

> If you are using a clock speed faster than the standard 100 kHz, you should ensure that no activity happens on the bus for 100 µs after the end of a block read command. Failure to do this could lead to I²C communication issues.

Clock stretching

The Tic uses a feature of I²C called *clock stretching*, meaning that it sometimes holds the SCL line low

to delay I²C communication while it is busy with other tasks and has not gotten around to processing data from the master. This means that the Tic is only compatible with I²C masters that also support clock stretching. It also means that the time to send an I²C command to the Tic is variable, even if you are only writing data and not reading anything.

The Tic only uses clock stretching at most once per byte. It stretches its clock after it receives a matching address byte, after it receives a command/data byte, and after it sends a byte that is not the last byte in a read transfer.

Each time the Tic stretches the clock, it could stretch the clock for as long as 1.5 ms if it is busy running its step planning algorithm, which it does approximately (but never more frequently than) every 5 ms. At other times, the Tic will usually stretch the clock for 150 μs or less, depending on the timing of the I²C bytes and how busy the Tic is performing other tasks.

The "serial response delay" setting can be used to guarantee a minimum amount of clock stretching each time the Tic stretches the clock.

11. USB command encoding

As described in **Section 8**, the Tic's USB commands each use one of four formats: **quick**, **7-bit write**, **32-bit write**, and **block read** (except for the **"set setting"** **[https://www.pololu.com/docs/0J71/8#cmd-set-setting]** command that uses its own unique format, also described below). This section explains how these four command formats are encoded as USB control transfer requests.

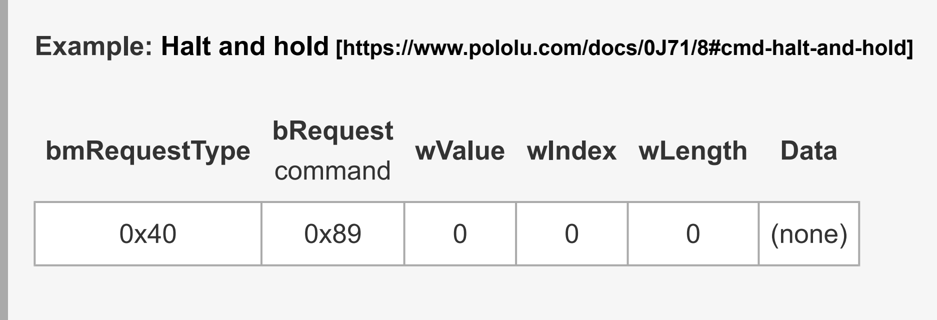
For a reference implementation of this protocol, see our **Tic Stepper Motor Controller software source code** **[https://github.com/pololu/pololu-tic-software]**.

All of the Tic's USB commands are implemented as vendor-defined control transfers on endpoint 0. The tables below show what data is sent in the SETUP packet and the data phase of the control transfer.

Quick

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0x40	command	0	0	0	(none)

A quick command sends no data and is a request that simply contains the command byte in the *bRequest* field.



Example: Halt and hold **[https://www.pololu.com/docs/0J71/8#cmd-halt-and-hold]**

bmRequestType	bRequest command	wValue	wIndex	wLength	Data
0x40	0x89	0	0	0	(none)

7-bit write

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0x40	command	data	0	0	(none)

A 7-bit write command is a request that contains the command byte in *bRequest* and a 7-bit value in the *wValue* field (the upper byte of *wValue* and the the most-significant bit of the lower byte are always 0).

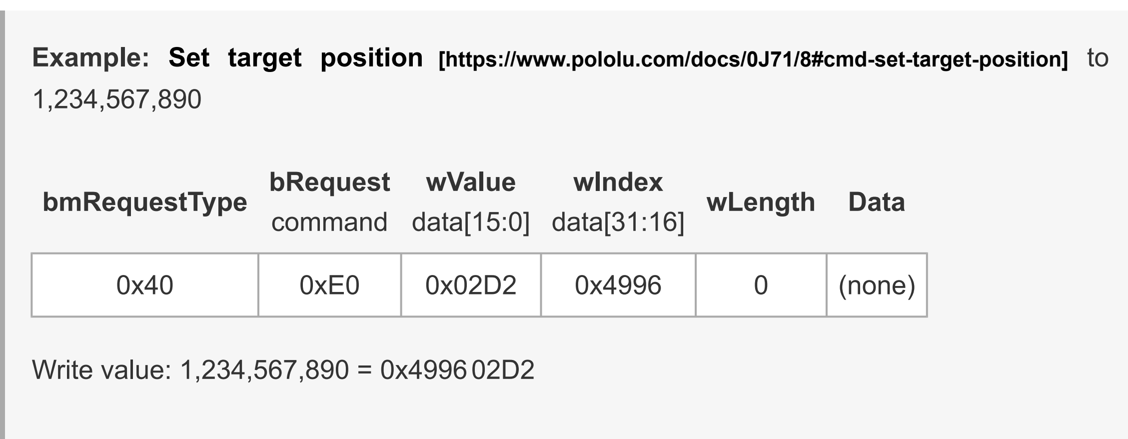
Example: Set step mode **[https://www.pololu.com/docs/0J71/8#cmd-set-step-mode]** to 1/8 step

bmRequestType	bRequest command	wValue data	wIndex	wLength	Data
0x40	0x94	0x0003	0	0	(none)

32-bit write

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0x40	command	data[15:0]	data[31:16]	0	(none)

A 32-bit write command is a request that contains the command byte in *bRequest* and a 32-bit value split between the *wValue* field (lower two bytes) and the *wIndex* field (upper two bytes).



Example: Set target position **[https://www.pololu.com/docs/0J71/8#cmd-set-target-position]** to 1,234,567,890

bmRequestType	bRequest command	wValue data[15:0]	wIndex data[31:16]	wLength	Data
0x40	0xE0	0x02D2	0x4996	0	(none)

Write value: 1,234,567,890 = 0x4996 02D2

Block read

bmRequestType	bRequest	wValue	wIndex	wLength	**Data** (in)
0xC0	command	0	offset	length	response data (*length* bytes)

A block read command reads a block of data from the Tic. The *bRequest* field holds the command byte, the offset in *wIndex* specifies the offset within the data that the response should start at, and *wLength* specifies how many bytes of data the response should include. The length must be between 1 and 128. Any multi-byte values contained in the response are in little-endian order (starting with the least-significant byte).

Example: Get variable [https://www.pololu.com/docs/0J71/8#cmd-get-variable] “target position” (32 bits)

bmRequestType	**bRequest** command	**wValue**	**wIndex** offset	**wLength** length	**Data** (in) response data
0xC0	0xA1	0	0x000A	0x0004	0x4996 02D2

Read value: 0x4996 02D2 = 1,234,567,890

Set setting

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**	**Data**
0x40	command	data	offset	0	(none)

The **“set setting” [https://www.pololu.com/docs/0J71/8#cmd-set-setting]** command uses a unique format. This request contains the command byte in *bRequest*, an 8-bit value in *wValue* (the upper byte is always 0), and an offset in *wIndex*.

Example: Set setting [https://www.pololu.com/docs/0J71/8#cmd-set-setting] “step mode” to 1/8 step

bmRequestType	**bRequest** command	**wValue** data	**wIndex** offset	**wLength**	**Data**
0x40	0x13	0x0003	0x0041	0	(none)

© 2001–2019 Pololu Corporation

12. Writing PC software to control the Tic

This section is about writing computer software to control the Tic.

Picking an interface

First, you should decide which interface of the Tic your software will talk to: serial, I²C, or USB. The USB interface is the most natural interface to use for PC software since most computers have USB ports, and we provide Tic software that uses the USB interface. However, if your computer has a serial or I²C port, or you want to control the Tic over a radio link, you might decide not to use the USB interface. All three of these interfaces support nearly the same set of commands, which are documented in **Section 8**. Most of this section is about using the USB interface.

Picking a USB API

If you decide to use the USB interface to communicate with the Tic, the next thing to do is decide which API you want to use to access it.

Almost every operating system has its own API for accessing USB devices, so you could use the native USB API of your operating system. You would need to decide which commands you are going to send to the Tic by reading **Section 8**, then figure out how to encode those commands for the Tic's USB interface by reading **Section 11**, and finally figure out how to send those USB commands using the API you have chosen by reading the documentation of that API. These APIs are typically meant for C programs, so you have to carefully manage pointers and memory allocation yourself. However, you might be able to find an API wrapper in the language of your choice that takes care of managing pointers for you.

Another option is to use a USB abstraction library like **libusbp [https://github.com/pololu/libusbp]** or **libusb [http://libusb.info/]**. These libraries abstract away the differences between USB APIs for the different operating systems they support, so you can write code that works on multiple operating systems. You would still have to read the documentation of the Tic's commands and USB protocol, read the documentation of the API you have chosen, and carefully manage pointers. However, you might be able to find a library wrapper in the language of your choice that manages pointers for you.

In our **Tic software package [https://github.com/pololu/pololu-tic-software]**, we provide a C library called *libpololu-tic* that uses libusbp to talk to the Tic. You can compile it as a static or shared library, and it has a stable C API. The library takes care of the details of encoding the Tic's USB commands so that you don't have to know much about the Tic's USB interface—you just have to call the appropriate function for each command you want to send. The library is designed to be used from C or C++, and you can access it from almost any other language if you write the appropriate foreign function interface (FFI) code. You would need to read the documentation of the library in the `include/tic.h` file to understand how to use the library, and carefully manage pointers.

In general, the easiest way to write software to control the Tic over USB is to install the Tic software and then invoke the Tic Command-line Utility (*ticcmd*), which is built on top of libpololu-tic. You can run `ticcmd` in a command prompt with no arguments to see what arguments the program supports. For example, to send an Exit Safe Start command and tell the Tic to travel to a particular position, you could run `ticcmd --exit-safe-start --position 1234` . If ticcmd is installed correctly, then its folder should be listed in your PATH environment variable, meaning that other programs can find it and run it without knowing exactly where it is. The ticcmd utility takes care of all the details of finding the Tic you want to talk to, encoding your command properly for the Tic’s USB interface, sending the command, and then cleaning up after itself. If an error happens, ticcmd will print an error message to its standard error pipe and return a non-zero exit code. Your software can look at the non-zero exit code to detect if an error happened, or just ignore the error. For more information about getting started with `ticcmd` , see **Section 4.4**.

Resources from the community

Please note that these resources are of varying quality and most are not tested or supported by Pololu.

- **pytic [https://gitlab.com/codebuk/pytic]**: Python code that uses PyUSB to access the Tic’s USB interface. It was written by customer Dan Tyrrell.
- **PyTic [https://github.com/AllenInstitute/pytic]**: A Python wrapper for libpololu-tic that provides access to most features of the C API. It was written by customer Daniel Castelli of the Allen Institute.
- **TicDotNet [https://github.com/jigarciacortazar/TicDotNet]**: A C# .NET example that uses LibUsbDotNet to access the Tic’s USB interface. It was written by customer jigarciacortazar.

12.1. Example code to run ticcmd in C

The example C code below shows how to invoke the Tic Command-line Utility (ticcmd) to control a Tic via USB. It demonstrates how to set the target position of the Tic using the `--position` option. This code works on Windows, Linux, and macOS.

The Tic’s control mode should be set to “Serial / I²C / USB”.

If you have multiple Tic devices connected to your computer via USB, you will need to use the `-d` option to specify the serial number of the device you want to use. For example, to set the target position to 200 on a Tic with serial number 00123456, you can run the command `ticcmd -d 00123456 --target 200` . You can run `ticcmd --list` in a shell to get the serial numbers of all the connected Tic devices.

If you want to set the target velocity instead of the target position, you can change the `--position` argument to `--velocity` . For a list of other options supported by the Tic Command-line Utility, run

`ticcmd` with no arguments.

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

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can run `ticcmd --reset-command-timeout` every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```
// Uses ticcmd to control the Tic over USB.
//
// NOTE: The Tic's control mode must be "Serial / I2C / USB".

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

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

// Sets the target position, returning 0 on success and -1 on failure.
int tic_set_target_position(int32_t target)
{
  char command[1024];
  snprintf(command, sizeof(command), "ticcmd --exit-safe-start --position %d", target);
  return run_command(command);
}

int main()
{
  printf("Setting target position to 800.\n");
  int result = tic_set_target_position(800);
  if (result) { return 1; }
  return 0;
}
```

12.2. Example code to run ticcmd in Ruby

The example Ruby code below shows how to invoke the Tic Command-line Utility (ticcmd) to send and receive data from a Tic via USB. It demonstrates how to set the target position of the Tic using the

`--position` option and how to read variables using the `-s` option. This code uses the YAML parser that comes with Ruby to parse the output of `ticcmd` when reading data from the Tic. This code should work on any system with ticcmd and Ruby.

The Tic's control mode should be set to "Serial / I²C / USB".

If you have multiple Tic devices connected to your computer via USB, you will need to use the `-d` option to specify the serial number of the device you want to use. For example, to set the target position to 200 on a Tic with serial number 00123456, you can run the command `ticcmd -d 00123456 --target 200`. You can run `ticcmd --list` in a shell to get the serial numbers of all the connected Tic devices.

If you want to set the target velocity instead of the target position, you can change the `--position` argument to `--velocity`. For a list of other options supported by the Tic Command-line Utility, run `ticcmd` with no arguments.

In the example below, the child ticcmd process uses the same error pipe as the Ruby process, so you will see any error messages printed by ticcmd if you run the Ruby program in a terminal. If you want to instead capture the standard error output so that you can use it in your Ruby program, you can change the code to use Ruby's **`Open3.capture3` method [http://ruby-doc.org/stdlib/libdoc/open3/rdoc/Open3.html#method-c-capture3]**.

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can run `ticcmd --reset-command-timeout` every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

© 2001–2019 Pololu Corporation

```
# Uses ticcmd to send and receive data from the Tic over USB.
#
# NOTE: The Tic's control mode must be "Serial / I2C / USB".

require 'open3'
require 'yaml'

def ticcmd(*args)
  command = 'ticcmd ' + args.join(' ')
  stdout, process_status = Open3.capture2(command)
  if !process_status.success?
    raise "Command failed with code #{process_status.exitstatus}: #{command}"
  end
  stdout
end

status = YAML.load(ticcmd('-s', '--full'))

position = status.fetch('Current position')
puts "Current position is #{position}."

new_target = position > 0 ? -200 : 200
puts "Setting target position to #{new_target}."
ticcmd('--exit-safe-start', '--position', new_target)
```

12.3. Example code to run ticcmd in Python

The example Python code below shows how to invoke the Tic Command-line Utility (ticcmd) to send and receive data from a Tic via USB. It demonstrates how to set the target position of the Tic using the `--position` option and how to read variables using the `-s` option. This code uses the **PyYAML [https://pyyaml.org/wiki/PyYAML]** library to parse the output of `ticcmd` when reading data from the Tic. This code should work on any system with ticcmd, Python, and PyYAML.

The Tic's control mode should be set to "Serial / I²C / USB".

If you have multiple Tic devices connected to your computer via USB, you will need to use the `-d` option to specify the serial number of the device you want to use. For example, to set the target position to 200 on a Tic with serial number 00123456, you can run the command `ticcmd -d 00123456 --target 200`. You can run `ticcmd --list` in a shell to get the serial numbers of all the connected Tic devices.

If you want to set the target velocity instead of the target position, you can change the `--position` argument to `--velocity`. For a list of other options supported by the Tic Command-line Utility, run `ticcmd` with no arguments.

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

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can run `ticcmd --reset-command-timeout` every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```
1  # Uses ticcmd to send and receive data from the Tic over USB.
2  # Works with either Python 2 or Python 3.
3  #
4  # NOTE: The Tic's control mode must be "Serial / I2C / USB".
5
6  import subprocess
7  import yaml
8
9  def ticcmd(*args):
10   return subprocess.check_output(['ticcmd'] + list(args))
11
12 status = yaml.load(ticcmd('-s', '--full'))
13
14 position = status['Current position']
15 print("Current position is {}.".format(position))
16
17 new_target = -200 if position > 0 else 200
18 print("Setting target position to {}.".format(new_target))
19 ticcmd('--exit-safe-start', '--position', str(new_target))
```

PyYAML installation tips

If you run the code above and get the error "ImportError: No module named yaml" or "ModuleNotFoundError: No module named 'yaml'", it means that the PyYAML library is not installed.

On Raspbian, Ubuntu, and other Debian-based operating systems, you can install PyYAML for both Python 2 and Python 3 by running this command:

```
sudo apt-get install python-yaml python3-yaml
```

Alternatively, if your system has the `pip`, `pip2`, or `pip3` command, you can use that to install the library. The correct command to use depends on how your system is set up and what version of Python you want to use, but it will most likely be one of the commands below:

```
pip install pyyaml
pip2 install pyyaml
pip3 install pyyaml
```

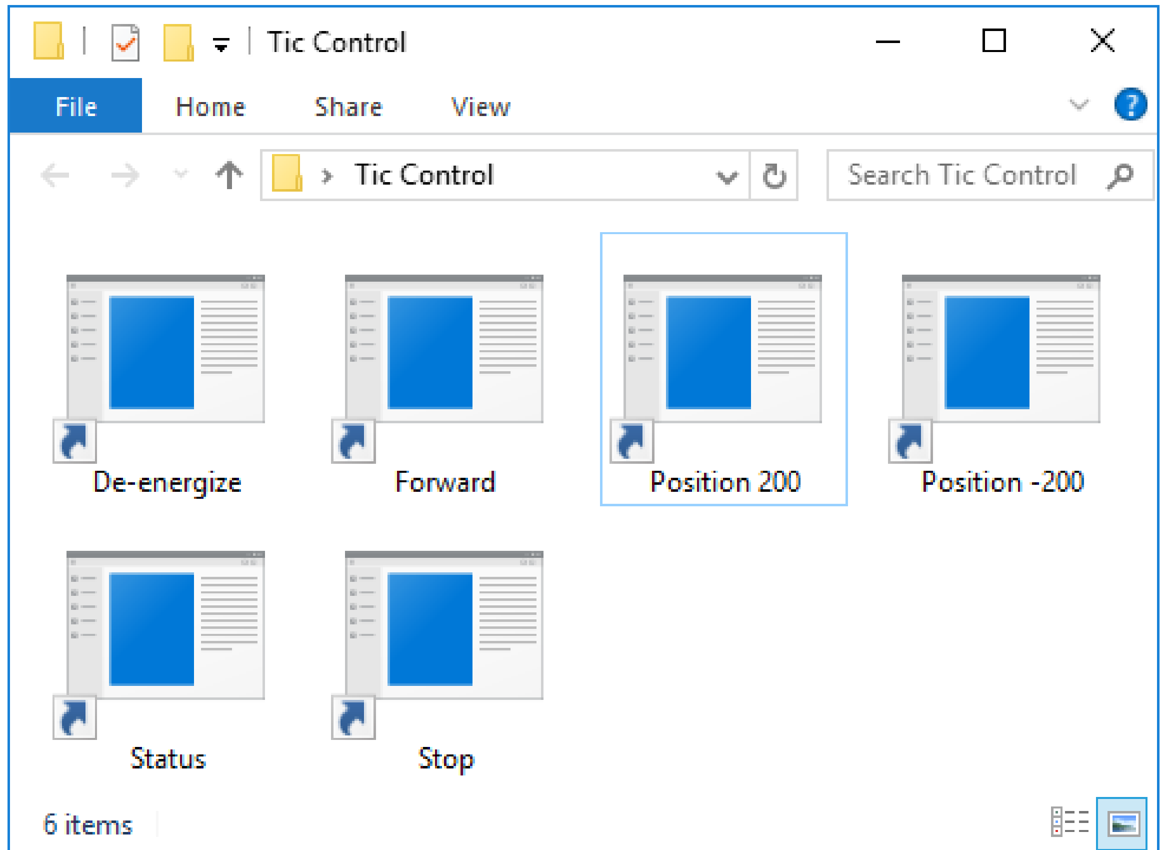
If your system does not have `pip`, `pip2`, or `pip3` (like macOS and MSYS2), you can install PyYAML from source by following the download and installation instructions on the **PyYAML [https://pyyaml.org/wiki/PyYAML]** web page.

 © 2001–2019 Pololu Corporation

Please note that PyYAML is only used for parsing the output of `ticcmd` in order to read data from the Tic. If you have trouble installing PyYAML but you do not actually need to read data from the Tic, you can simply remove the code that uses it. Also, the output of `ticcmd` is simple enough that you might consider writing your own functions to extract data from it instead of relying on a third-party library.

12.4. Running ticcmd with Windows shortcuts

This section explains how to control the Tic with Windows shortcuts. A folder of shortcuts can serve as a simple GUI for controlling the Tic and requires no programming experience to make. This section focuses on shortcuts in Windows, but you should be able to do something similar in Linux or macOS.



A folder with shortcuts to ticcmd that could be used for controlling a Tic.

The Tic Command-line Utility (ticcmd) provides many commands for controlling the Tic, and gets installed along with the rest of the Tic software and drivers. After installing ticcmd as described in **Section 3.1**, you can follow these instructions to make a shortcut to it:

1. Right-click on the blank area of your desktop or any other folder, open the "New" menu, and select "Shortcut".

2. Windows will open a "Create Shortcut" wizard and prompt you for the location of the item you want to make a shortcut to. Type `ticcmd --resume --position 200 --pause-on-error` or any other command you want to run.
3. Click "Next". If Windows says "The file ticcmd cannot be found.", it either means you have not installed the Tic software, or the folder containing `ticcmd.exe` is not listed in your PATH environment variable (maybe you unchecked the checkbox that adds it your PATH when installing the software). As a solution, you could replace `ticcmd` in the input box with the full path to `ticcmd.exe` in double quotes. However, the easiest solution is probably to reinstall the Tic software, making sure to leave the "Add the bin directory to the PATH environment variable." checkbox checked this time. You might also need to restart your computer.
4. Type an appropriate name for your shortcut and click "Finish".

Now you should be able to double-click on your shortcut to run the command you entered. You will probably see a black window briefly flash on the screen run the command runs.

At this point, you might get an error message that says:

> *Error: Failed to open generic handle. Access is denied. Try closing all other programs that are using the device. Windows error code 0x5.*

It is best to follow the advice in the error message and close the Tic Control Center and any other programs that might be using the Tic. In Windows, only one program can use the Tic's USB interface at a time.

Another issue you might notice is that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

Example commands

Here are some example commands you might want to put into a shortcut, along with information about how the command works:

```
ticcmd --resume --position 200 --pause-on-error
```

The command above will send the "Energize" and "Exit safe start" commands and set the Tic's target position 200. The Tic's input mode should be set to "Serial / I²C / USB" for this command to work. The `--pause-on-error` option means that if there are any errors communicating with the Tic, then ticcmd

will keep running until you press Enter or close the window. This prevents the ticcmd output window from closing immediately and allows you to read the error message. However, if there are some errors that are still happening on the Tic that prevent the motor from moving, there will be no notification.

```
ticcmd --resume --velocity 1000000 --pause-on-error
```

The command above will send the "Energize" and "Exit safe start" commands and set the Tic's target velocity to 1000000 (100 pulses per second). You can set the velocity to 0 if you want the Tic to decelerate to a stop, or set it to a negative number to make it turn the other way. As in the previous command, `--pause-on-error` gives you a chance to see error messages if there is any problem communicating with the Tic.

```
ticcmd --deenergize --pause-on-error
```

The command above will send the "De-energize" command, causing the Tic to de-energize its stepper motor coils by disabling its stepper motor driver. You can find more details about this command in **Section 8**. As in the previous command, `--pause-on-error` gives you a chance to see error messages if there is any problem communicating with the Tic.

```
ticcmd --status --pause
```

The command above shows status information from the Tic. You can add the `--full` option to see more information. The `--pause` option tells ticcmd to always wait for you to press Enter or close the window before terminating, which gives you a chance to see the output. By default, the ticcmd console window will not have enough lines to show the full status output all at once. To fix this, you can modify the shortcut to make the console have more lines: right-click on the shortcut, select "Properties", select the "Layout" tab, and then in the "Window Size" box change the "Height".

Multiple Tic devices

If you have multiple Tic devices connected to the computer via USB, you will need to add the `-d` option to each shortcut in order to specify the serial number of the device you want to use. For example, a shortcut that gets the status of the Tic with serial number 12345678 would have a command like `ticcmd -d 12345678 --status --pause`. You can see your Tic's serial number in the Tic Control Center or by running `ticcmd --list`.

Multiple commands in one shortcut

You can specify multiple commands to the Tic in one shortcut by just adding more options. For example, you could temporarily change the current limit and set the target position at the same time with a command like `ticcmd --current 500 --position 1234 --pause-on-error`. The order of the options does not matter; the utility performs the commands in a predetermined order. If you need something a bit more complicated, you might consider writing a Batch or PowerShell script.

Shortcut customizations

In the Properties dialog for a shortcut, you can change its icon or assign a shortcut key. You can add shortcuts to your Start Menu or make a toolbar of shortcuts inside the taskbar for quick access.

12.5. Example serial code for Linux and macOS in C

The example C code below uses parts of the POSIX API provided by Linux and macOS to communicate with a Tic via serial. It demonstrates how to set the target position of the Tic and how to read variables from it. For a very similar example that works on Windows, see **Section 12.6**.

The Tic's control mode should be set to "Serial / I²C / USB".

You will need to change the `const char * device` line in the code in order to specify the correct serial port. The correct serial port name to use depends on your operating system and what type of serial port or USB-to-serial adapter you are using between the Tic and your computer.

The baud rate you select in the code should match the baud rate specified in the serial settings in the Tic Control Center. The code below uses 9600 baud, but you can easily change it to use any of the following standard baud rates: 4800, 9600, 19200, 38400, or 115200. Due to hardware limitations, the Tic cannot exactly produce 38400 baud or 115200 baud, but it can use similar baud rates that are close enough to work.

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a "Reset command timeout" command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```c
// Uses POSIX serial port functions to send and receive data from a Tic.
// NOTE: The Tic's control mode must be "Serial / I2C / USB".
// NOTE: You will need to change the 'const char * device' line below to
//   specify the correct serial port.

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

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

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

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

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

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

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

© 2001–2019 Pololu Corporation

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

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

  return fd;
}

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

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

// Sends the "Exit safe start" command.
// Returns 0 on success and -1 on failure.
int tic_exit_safe_start(int fd)
{
  uint8_t command[] = { 0x83 };
  return write_port(fd, command, sizeof(command));
}
```

```

// Sets the target position, returning 0 on success and -1 on failure.
//
// For more information about what this command does, see the
// "Set target position" command in the "Command reference" section of the
// Tic user's guide.
int tic_set_target_position(int fd, int32_t target)
{
  uint32_t value = target;
  uint8_t command[6];
  command[0] = 0xE0;
  command[1] = ((value >>  7) & 1) |
               ((value >> 14) & 2) |
               ((value >> 21) & 4) |
               ((value >> 28) & 8);
  command[2] = value >> 0 & 0x7F;
  command[3] = value >> 8 & 0x7F;
  command[4] = value >> 16 & 0x7F;
  command[5] = value >> 24 & 0x7F;
  return write_port(fd, command, sizeof(command));
}

// Gets one or more variables from the Tic.
// Returns 0 for success, -1 for failure.
int tic_get_variable(int fd, uint8_t offset, uint8_t * buffer, uint8_t length)
{
  uint8_t command[] = { 0xA1, offset, length };
  int result = write_port(fd, command, sizeof(command));
  if (result) { return -1; }
  ssize_t received = read_port(fd, buffer, length);
  if (received < 0) { return -1; }
  if (received != length)
  {
    fprintf(stderr, "read timeout: expected %u bytes, got %zu\n",
      length, received);
    return -1;
  }
  return 0;
}

// Gets the "Current position" variable from the Tic.
// Returns 0 for success, -1 for failure.
int tic_get_current_position(int fd, int32_t * output)
{
  *output = 0;
  uint8_t buffer[4];
  int result = tic_get_variable(fd, 0x22, buffer, sizeof(buffer));
  if (result) { return -1; }
  *output = buffer[0] + ((uint32_t)buffer[1] << 8) +
    ((uint32_t)buffer[2] << 16) + ((uint32_t)buffer[3] << 24);
  return 0;
}

int main()
{
  // Choose the serial port name.
  const char * device = "/dev/ttyACM0";

  // Choose the baud rate (bits per second).  This must match the baud rate in
  // the Tic's serial settings.
  uint32_t baud_rate = 9600;

```

```
187   int fd = open_serial_port(device, baud_rate);
188   if (fd < 0) { return 1; }
189
190   int result;
191
192   int32_t position;
193   result = tic_get_current_position(fd, &position);
194   if (result) { return 1; }
195   printf("Current position is %d.\n", position);
196
197   int32_t new_target = position > 0 ? -200 : 200;
198   printf("Setting target position to %d.\n", new_target);
199   result = tic_exit_safe_start(fd);
200   if (result) { return 1; }
201   result = tic_set_target_position(fd, new_target);
202   if (result) { return 1; }
203
204   close(fd);
205   return 0;
206 }
```

12.6. Example serial code for Windows in C

The example C code below uses the Windows API to communicate with a Tic via serial. It demonstrates how to set the target position of the Tic and how to read variables from it. For a very similar example that works on Linux and macOS, see **Section 12.5**.

The Tic's control mode should be set to "Serial / I²C / USB".

You will need to change the `const char * device` line in the code in order to specify the correct serial port. The correct serial port name to use depends on your operating system and what type of serial port or USB-to-serial adapter you are using between the Tic and your computer.

The baud rate you select in the code should match the baud rate specified in the serial settings in the Tic Control Center.

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a "Reset command timeout" command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```c
// Uses Windows API serial port functions to send and receive data from a Tic.
// NOTE: The Tic's control mode must be "Serial / I2C / USB".
// NOTE: You will need to change the 'const char * device' line below to
//   specify the correct serial port.

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

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

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

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

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

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

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

```
63    {
64      print_error("Failed to get serial settings");
65      CloseHandle(port);
66      return INVALID_HANDLE_VALUE;
67    }
68
69    state.BaudRate = baud_rate;
70
71    success = SetCommState(port, &state);
72    if (!success)
73    {
74      print_error("Failed to set serial settings");
75      CloseHandle(port);
76      return INVALID_HANDLE_VALUE;
77    }
78
79    return port;
80  }
81
82  // Writes bytes to the serial port, returning 0 on success and -1 on failure.
83  int write_port(HANDLE port, uint8_t * buffer, size_t size)
84  {
85    DWORD written;
86    BOOL success = WriteFile(port, buffer, size, &written, NULL);
87    if (!success)
88    {
89      print_error("Failed to write to port");
90      return -1;
91    }
92    if (written != size)
93    {
94      print_error("Failed to write all bytes to port");
95      return -1;
96    }
97    return 0;
98  }
99
100 // Reads bytes from the serial port.
101 // Returns after all the desired bytes have been read, or if there is a
102 // timeout or other error.
103 // Returns the number of bytes successfully read into the buffer, or -1 if
104 // there was an error reading.
105 SSIZE_T read_port(HANDLE port, uint8_t * buffer, size_t size)
106 {
107   DWORD received;
108   BOOL success = ReadFile(port, buffer, size, &received, NULL);
109   if (!success)
110   {
111     print_error("Failed to read from port");
112     return -1;
113   }
114   return received;
115 }
116
117 // Sends the "Exit safe start" command.
118 // Returns 0 on success and -1 on failure.
119 int tic_exit_safe_start(HANDLE port)
120 {
121   uint8_t command[] = { 0x83 };
122   return write_port(port, command, sizeof(command));
123 }
124
```

```
125 // Sets the target position, returning 0 on success and -1 on failure.
126 //
127 // For more information about what this command does, see the
128 // "Set target position" command in the "Command reference" section of the
129 // Tic user's guide.
130 int tic_set_target_position(HANDLE port, int32_t target)
131 {
132   uint32_t value = target;
133   uint8_t command[6];
134   command[0] = 0xE0;
135   command[1] = ((value >>  7) & 1) |
136                ((value >> 14) & 2) |
137                ((value >> 21) & 4) |
138                ((value >> 28) & 8);
139   command[2] = value >> 0 & 0x7F;
140   command[3] = value >> 8 & 0x7F;
141   command[4] = value >> 16 & 0x7F;
142   command[5] = value >> 24 & 0x7F;
143   return write_port(port, command, sizeof(command));
144 }
145
146 // Gets one or more variables from the Tic.
147 // Returns 0 for success, -1 for failure.
148 int tic_get_variable(HANDLE port, uint8_t offset,
149   uint8_t * buffer, uint8_t length)
150 {
151   uint8_t command[] = { 0xA1, offset, length };
152   int result = write_port(port, command, sizeof(command));
153   if (result) { return -1; }
154   SSIZE_T received = read_port(port, buffer, length);
155   if (received < 0) { return -1; }
156   if (received != length)
157   {
158     fprintf(stderr, "read timeout: expected %u bytes, got %ld\n",
159       length, received);
160     return -1;
161   }
162   return 0;
163 }
164
165 // Gets the "Current position" variable from the Tic.
166 // Returns 0 for success, -1 for failure.
167 int tic_get_current_position(HANDLE port, int32_t * output)
168 {
169   *output = 0;
170   uint8_t buffer[4];
171   int result = tic_get_variable(port, 0x22, buffer, sizeof(buffer));
172   if (result) { return -1; }
173   *output = buffer[0] + ((uint32_t)buffer[1] << 8) +
174     ((uint32_t)buffer[2] << 16) + ((uint32_t)buffer[3] << 24);
175   return 0;
176 }
177
178 int main()
179 {
180   // Choose the serial port name.
181   const char * device = "\\\\.\\COM7";
182
183   // Choose the baud rate (bits per second).  This must match the baud rate in
184   // the Tic's serial settings.
185   uint32_t baud_rate = 9600;
186
```

```
187     HANDLE port = open_serial_port(device, baud_rate);
188     if (port == INVALID_HANDLE_VALUE) { return 1; }
189
190     int result;
191
192     int32_t position;
193     result = tic_get_current_position(port, &position);
194     if (result) { return 1; }
195     printf("Current position is %d.\n", position);
196
197     int32_t new_target = position > 0 ? -200 : 200;
198     printf("Setting target position to %d.\n", new_target);
199     result = tic_exit_safe_start(port);
200     if (result) { return 1; }
201     result = tic_set_target_position(port, new_target);
202     if (result) { return 1; }
203
204     CloseHandle(port);
205     return 0;
206   }
```

12.7. Example serial code in Python

The example Python code below uses the **pySerial** **[http://pyserial.readthedocs.io]** library to communicate with the Tic via serial. It demonstrates how to set the target position of the Tic and how to read variables from it.

The Tic's control mode should be set to "Serial / I²C / USB".

You will need to change the line that sets `port_name` in order to specify the correct serial port. The correct serial port name to use depends on your operating system and what type of serial port or USB-to-serial adapter you are using between the Tic and your computer. The baud rate you select in the code should match the baud rate specified in the serial settings in the Tic Control Center.

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

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a "Reset command timeout" command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```python
# Uses the pySerial library to send and receive data from a Tic.
#
# NOTE: The Tic's control mode must be "Serial / I2C / USB".
# NOTE: You will need to change the "port_name =" line below to specify the
#   right serial port.

import serial

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

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

  # Sends the "Exit safe start" command.
  def exit_safe_start(self):
    self.send_command(0x83)

  # Sets the target position.
  #
  # For more information about what this command does, see the
  # "Set target position" command in the "Command reference" section of the
  # Tic user's guide.
  def set_target_position(self, target):
    self.send_command(0xE0,
     ((target >>  7) & 1) | ((target >> 14) & 2) |
     ((target >> 21) & 4) | ((target >> 28) & 8),
     target >> 0 & 0x7F,
     target >> 8 & 0x7F,
     target >> 16 & 0x7F,
     target >> 24 & 0x7F)

  # Gets one or more variables from the Tic.
  def get_variables(self, offset, length):
    self.send_command(0xA1, offset, length)
    result = self.port.read(length)
    if len(result) != length:
      raise RuntimeError("Expected to read {} bytes, got {}."
        .format(length, len(result)))
    return bytearray(result)

  # Gets the "Current position" variable from the Tic.
  def get_current_position(self):
    b = self.get_variables(0x22, 4)
    position = b[0] + (b[1] << 8) + (b[2] << 16) + (b[3] << 24)
    if position >= (1 << 31):
      position -= (1 << 32)
    return position

# Choose the serial port name.
port_name = "/dev/ttyACM0"

# Choose the baud rate (bits per second).  This must match the baud rate in
# the Tic's serial settings.
baud_rate = 9600

```

```
63 # Change this to a number between 0 and 127 that matches the device number of
64 # your Tic if there are multiple serial devices on the line and you want to
65 # use the Pololu Protocol.
66 device_number = None
67
68 port = serial.Serial(port_name, baud_rate, timeout=0.1, write_timeout=0.1)
69
70 tic = TicSerial(port, device_number)
71
72 position = tic.get_current_position()
73 print("Current position is {}.".format(position))
74
75 new_target = -200 if position > 0 else 200
76 print("Setting target position to {}.".format(new_target));
77 tic.exit_safe_start()
78 tic.set_target_position(new_target)
```

12.8. Example I²C code for Linux in C

The example C code below uses the I²C API provided by the Linux kernel to send and receive data from a Tic. It demonstrates how to set the target position of the Tic and how to read variables from it. This code only works on Linux.

If you are using a Raspberry Pi, please note that the Raspberry Pi's hardware I²C module has a **bug [https://github.com/raspberrypi/linux/issues/254]** that causes this code to not work reliably. As a workaround, we recommend enabling the i2c-gpio overlay and using the I²C device that it provides. To do this, add the line `dtoverlay=i2c-gpio` to `/boot/config.txt` and reboot. The **overlay documentation [https://github.com/raspberrypi/firmware/tree/master/boot/overlays]** has information about the parameters you can put on that line, but those parameters are not required. Connect the Tic's SDA line to GPIO23 and connect the Tic's SCL line to GPIO24. The i2c-gpio overlay creates a new I²C device which is usually named `/dev/i2c-3` , and the code below uses that device. To give your user permission to access I²C busses without being root, you might have to add yourself to the `i2c` group by running `sudo usermod -a -G i2c $(whoami)` and restarting.

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a "Reset command timeout" command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

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

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

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

// Sends the "Exit safe start" command.
// Returns 0 on success and -1 on failure.
int tic_exit_safe_start(int fd, uint8_t address)
{
  uint8_t command[] = { 0x83 };
  struct i2c_msg message = { address, 0, sizeof(command), command };
  struct i2c_rdwr_ioctl_data ioctl_data = { &message, 1 };
  int result = ioctl(fd, I2C_RDWR, &ioctl_data);
  if (result != 1)
  {
    perror("failed to exit safe start");
    return -1;
  }
  return 0;
}

// Sets the target position, returning 0 on success and -1 on failure.
//
// For more information about what this command does, see the
// "Set Target Position" command in the "Command reference" section of the
// Tic user's guide.
int tic_set_target_position(int fd, uint8_t address, int32_t target)
{
  uint8_t command[] = {
    0xE0,
    (uint8_t)(target >> 0  & 0xFF),
    (uint8_t)(target >> 8  & 0xFF),
    (uint8_t)(target >> 16 & 0xFF),
    (uint8_t)(target >> 24 & 0xFF),
  };
  struct i2c_msg message = { address, 0, sizeof(command), command };
  struct i2c_rdwr_ioctl_data ioctl_data = { &message, 1 };
```

```
  int result = ioctl(fd, I2C_RDWR, &ioctl_data);
  if (result != 1)
  {
    perror("failed to set target position");
    return -1;
  }
  return 0;
}

// Gets one or more variables from the Tic (without clearing them).
// Returns 0 for success, -1 for failure.
int tic_get_variable(int fd, uint8_t address, uint8_t offset,
  uint8_t * buffer, uint8_t length)
{
  uint8_t command[] = { 0xA1, offset };
  struct i2c_msg messages[] = {
    { address, 0, sizeof(command), command },
    { address, I2C_M_RD, length, buffer },
  };
  struct i2c_rdwr_ioctl_data ioctl_data = { messages, 2 };
  int result = ioctl(fd, I2C_RDWR, &ioctl_data);
  if (result != 2)
  {
    perror("failed to get variables");
    return -1;
  }
  return 0;
}

// Gets the "Current position" variable from the Tic.
// Returns 0 for success, -1 for failure.
int tic_get_current_position(int fd, uint8_t address, int32_t * output)
{
  *output = 0;
  uint8_t buffer[4];
  int result = tic_get_variable(fd, address, 0x22, buffer, sizeof(buffer));
  if (result) { return -1; }
  *output = buffer[0] + ((uint32_t)buffer[1] << 8) +
    ((uint32_t)buffer[2] << 16) + ((uint32_t)buffer[3] << 24);
  return 0;
}

int main()
{
  // Choose the I2C device.
  const char * device = "/dev/i2c-3";

  // Set the I2C address of the Tic (the device number).
  const uint8_t address = 14;

  int fd = open_i2c_device(device);
  if (fd < 0) { return 1; }

  int result;

  int32_t position;
  result = tic_get_current_position(fd, address, &position);
  if (result) { return 1; }
  printf("Current position is %d.\n", position);

  int32_t new_target = position > 0 ? -200 : 200;
  printf("Setting target position to %d.\n", new_target);
```

```
125      result = tic_exit_safe_start(fd, address);
126      if (result) { return 1; }
127      result = tic_set_target_position(fd, address, new_target);
128      if (result) { return 1; }
129
130      close(fd);
131      return 0;
132  }
```

12.9. Example I²C code for Linux in Python

The example Python code below uses the **smbus2** **[https://github.com/kplindegaard/smbus2]** library to send and receive data from a Tic via I²C. It demonstrates how to set the target position of the Tic and how to read variables from it. This example works on Linux with either Python 2 or Python 3.

If you are using a Raspberry Pi, please see the notes about setting up I²C for the Raspberry Pi in **Section 12.8**.

If you run the code and get the error "ImportError: No module named smbus2" or "ModuleNotFoundError: No module named 'smbus2'", it means that the smbus2 library is not installed, and you should follow the instructions on the **smbus2** **[https://github.com/kplindegaard/smbus2]** web page to install it.

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a "Reset command timeout" command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```python
# Uses the smbus2 library to send and receive data from a Tic.
# Works on Linux with either Python 2 or Python 3.
#
# NOTE: The Tic's control mode must be "Serial / I2C / USB".
# NOTE: For reliable operation on a Raspberry Pi, enable the i2c-gpio
#   overlay and use the I2C device it provides (usually /dev/i2c-3).
# NOTE: You might nee to change the 'SMBus(3)' line below to specify the
#   correct I2C device.
# NOTE: You might need to change the 'address = 11' line below to match
#   the device number of your Tic.

from smbus2 import SMBus, i2c_msg

class TicI2C(object):
  def __init__(self, bus, address):
    self.bus = bus
    self.address = address

  # Sends the "Exit safe start" command.
  def exit_safe_start(self):
    command = [0x83]
    write = i2c_msg.write(self.address, command)
    self.bus.i2c_rdwr(write)

  # Sets the target position.
  #
  # For more information about what this command does, see the
  # "Set target position" command in the "Command reference" section of the
  # Tic user's guide.
  def set_target_position(self, target):
    command = [0xE0,
      target >> 0 & 0xFF,
      target >> 8 & 0xFF,
      target >> 16 & 0xFF,
      target >> 24 & 0xFF]
    write = i2c_msg.write(self.address, command)
    self.bus.i2c_rdwr(write)

  # Gets one or more variables from the Tic.
  def get_variables(self, offset, length):
    write = i2c_msg.write(self.address, [0xA1, offset])
    read = i2c_msg.read(self.address, length)
    self.bus.i2c_rdwr(write, read)
    return list(read)

  # Gets the "Current position" variable from the Tic.
  def get_current_position(self):
    b = self.get_variables(0x22, 4)
    position = b[0] + (b[1] << 8) + (b[2] << 16) + (b[3] << 24)
    if position >= (1 << 31):
      position -= (1 << 32)
    return position

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

# Select the I2C address of the Tic (the device number).
address = 14

tic = TicI2C(bus, address)

position = tic.get_current_position()
```

```
63  print("Current position is {}.".format(position))
64
65  new_target = -200 if position > 0 else 200
66  print("Setting target position to {}.".format(new_target));
67  tic.exit_safe_start()
68  tic.set_target_position(new_target)
```

12.10. Example code using the C API

The code below uses the C API provided by the **Tic software** **[https://github.com/pololu/pololu-tic-software]** to send and receive data from a Tic over USB. This code is written in C and works on Windows, Linux, and macOS. For a very similar example using the C++ API, see **Section 12.11**.

If you have compiled the Tic software from source and installed it on a Unix-like system (including MSYS2 on Windows), and copied the code below to a file named `code.c`, you should be able to compile the code below by running this command in a shell:

```
gcc code.c $(pkg-config libpololu-tic-1 --cflags --libs)
```

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic's command timeout feature: by default, the Tic's "Command timeout" error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a "Reset command timeout" command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the "Enable command timeout" checkbox in the "Serial" box.

```
// Uses the Tic's C API to send and receive data from a Tic.
// NOTE: The Tic's control mode must be "Serial / I2C / USB".

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <tic.h>

bool handle_error(tic_error * error)
{
  if (error == NULL) { return false; }
  fprintf(stderr, "Error: %s\n", tic_error_get_message(error));
  tic_error_free(error);
  return true;
}

// Opens a handle to a Tic that can be used for communication.
//
// To open a handle to any Tic:
//   tic_handle * handle = open_handle(NULL);
// To open a handle to the Tic with serial number 01234567:
//   tic_handle * handle = open_handle("01234567");
tic_handle * open_handle(const char * desired_serial_number)
{
  tic_handle * handle = NULL;

  // Get a list of Tic devices connected via USB.
  tic_device ** list = NULL;
  size_t count = 0;
  tic_error * error = tic_list_connected_devices(&list, &count);
  if (handle_error(error)) { goto cleanup; }

  // Iterate through the list and select one device.
  tic_device * device = NULL;
  for (size_t i = 0; i < count; i++)
  {
    tic_device * candidate = list[i];

    if (desired_serial_number)
    {
      const char * serial_number = tic_device_get_serial_number(candidate);
      if (strcmp(serial_number, desired_serial_number))
      {
        // Found a device with the wrong serial number, so continue on to
        // the next device in the list.
        continue;
      }
    }

    // Select this device as the one we want to connect to, and break
    // out of the loop.
    device = candidate;
    break;
  }

  if (device == NULL)
  {
    fprintf(stderr, "Error: No device found.\n");
    goto cleanup;
  }

  error = tic_handle_open(device, &handle);
```

```
63    if (handle_error(error)) { goto cleanup; }
64
65  cleanup:
66    for (size_t i = 0; i < count; i++)
67    {
68      tic_device_free(list[i]);
69    }
70    tic_list_free(list);
71    return handle;
72  }
73
74  int main()
75  {
76    int exit_code = 1;
77    tic_handle * handle = NULL;
78    tic_variables * variables = NULL;
79
80    handle = open_handle(NULL);
81    if (handle == NULL) { goto cleanup; }
82
83    tic_error * error = tic_get_variables(handle, &variables, false);
84    if (handle_error(error)) { goto cleanup; }
85
86    int32_t position = tic_variables_get_current_position(variables);
87    printf("Current position is %d.\n", position);
88
89    int32_t new_target = position > 0 ? -200 : 200;
90    printf("Setting target position to %d.\n", new_target);
91
92    error = tic_exit_safe_start(handle);
93    if (handle_error(error)) { goto cleanup; }
94
95    error = tic_set_target_position(handle, new_target);
96    if (handle_error(error)) { goto cleanup; }
97
98    exit_code = 0;  // This program ran successfully.
99
100 cleanup:
101   // Free the resources used by the variables.
102   tic_variables_free(variables);
103
104   // Call tic_handle_close() to free its resources and because Windows only
105   // allows one open handle per device.
106   // (Though, in this program, we are about to return from main, so the program
107   // will exit and the operating system will do this for us very soon.)
108   tic_handle_close(handle);
109
110   return exit_code;
111 }
```

12.11. Example code using the C++ API

The code below uses the C++ API provided by the **Tic software** **[https://github.com/pololu/pololu-tic-software]** to send and receive data from a Tic over USB. This code is written in C++ and works on Windows, Linux, and macOS. The code here is simpler than the equivalent C example in **Section 12.10** because the Tic's C++ API automatically frees allocated resources and converts errors into exceptions.

If you have compiled the Tic software from source and installed it on a Unix-like system (including MSYS2 on Windows), and copied the code below to a file named `code.cpp` , you should be able to compile the code below by running this command in a shell:

```
g++ code.cpp $(pkg-config libpololu-tic-1 --cflags --libs)
```

You might notice that the Tic only performs the desired movement for about a second before it stops moving and the red LED turns on, indicating an error. This is because of the Tic’s command timeout feature: by default, the Tic’s “Command timeout” error will happen if it does not receive certain commands periodically (see **Section 5.4** for details), causing the motor to stop. You can send a “Reset command timeout” command every second to get around this, or you can disable the command timeout feature using the Tic Control Center: uncheck the “Enable command timeout” checkbox in the “Serial” box.

```
// Uses the Tic's C++ API to send and receive data from a Tic.
// NOTE: The Tic's control mode must be "Serial / I2C / USB".

#include <iostream>
#include <tic.hpp>

// Opens a handle to a Tic that can be used for communication.
//
// To open a handle to any Tic:
//   tic_handle * handle = open_handle();
// To open a handle to the Tic with serial number 01234567:
//   tic_handle * handle = open_handle("01234567");
tic::handle open_handle(const char * desired_serial_number = nullptr)
{
  // Get a list of Tic devices connected via USB.
  std::vector<tic::device> list = tic::list_connected_devices();

  // Iterate through the list and select one device.
  for (const tic::device & device : list)
  {
    if (desired_serial_number &&
      device.get_serial_number() != desired_serial_number)
    {
      // Found a device with the wrong serial number, so continue on to
      // the next device in the list.
      continue;
    }

    // Open a handle to this device and return it.
    return tic::handle(device);
  }

  throw std::runtime_error("No device found.");
}

int main()
{
  try
  {
    tic::handle handle = open_handle();

    tic::variables vars = handle.get_variables();

    int32_t position = vars.get_current_position();
    std::cout << "Current position is " << position << ".\n";

    int32_t new_target = position > 0 ? -200 : 200;
    std::cout << "Setting target position to " << new_target << ".\n";

    handle.exit_safe_start();
    handle.set_target_position(new_target);
  }
  catch (const std::exception & error)
  {
    std::cerr << "Error: " << error.what() << std::endl;
    return 1;
  }
  return 0;
}
```