Instructions for Calibration of X-Carve

To calibrate the X-Carve you must open Universal Gcode Sender or UGS for short. There is a desktop shortcut on the main computer for carving with the java icon, but if using a different computer or if unavailable, download the newest available version from:

https://winder.github.io/ugs_website/download/

If link is dead, just look up Universal Gcode Sender online and download it.

Make sure the X-Carve is on and connected to the computer you are using via USB.

Select the COM port, there should only be one machine plugged in so there should only be one option available. For baud, select 115200. Make sure GRBL is selected as the firmware and then click open. After a few seconds you should see a message that looks like

Grbl 1.1f ['$' for help]

when this message appears you know you are connected to the X-Carve.

For instructions on using Grbl visit pages 3 - 13

To start calibration, go to the commands tab in UGS and type in $$ this will open a list of Grbl settings.

To see the meanings of all of these settings visit pages 14 - 23

The settings we will be focusing our attention on are the $100, $101, and $102 settings. These control the # of steps made by the stepper motor to move one mm on each axis (X, Y, and Z) respectively. To adjust any of these settings use the command line:

$(setting number)=value

for example: $101=40.

This command would make the steps per mm on the y axis equal to 40.

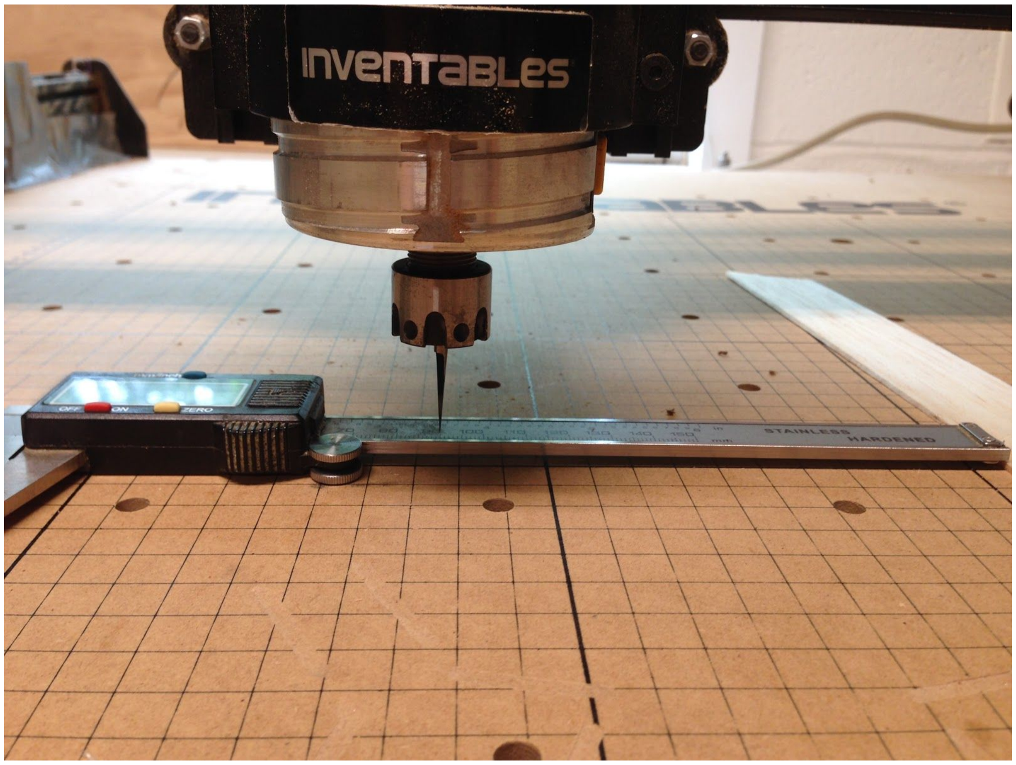
Calibration:

To calibrate the machine good starting values for steps/mm on each axis are:

X: 40
Y: 40
Z: 188.947.

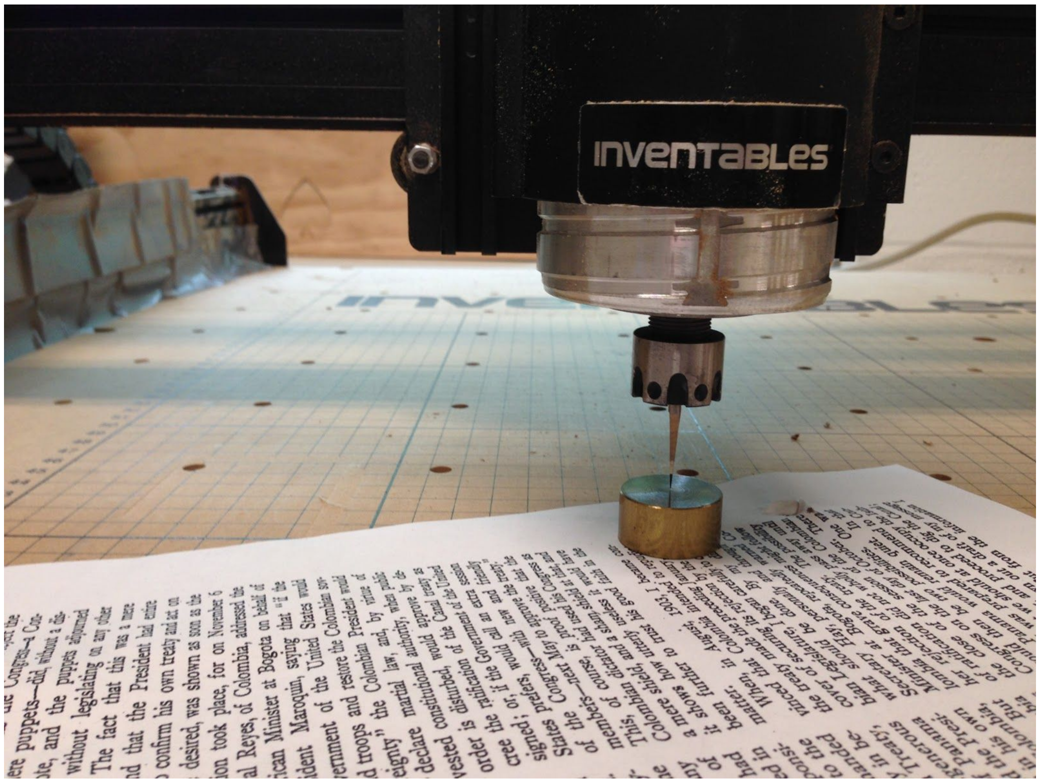
X and Y axes:

To start get a caliper with mm markings and and thin piece of material to keep caliper level with cutting surface. Align caliper with one of the lines from the cutting surface. Replace current bit with pointed bit to get exact measurements. Hone the bit along one of the mm marking on the caliper exactly using the machine controls in UGS. In the box labeled step size, enter a certain distance for the bit to move along the caliper, this value should be large enough to be able to determine the margin of error, but small enough to stay on the caliper, For example: 50 mm (Only move the bit this distance once). Measure the actual distance traveled. Divide the distance you entered by the distance actually traveled and multiply this value by the steps/mm on that axis (the value for $100 or $101 depending on axis). Once you have this value change the steps/mm setting of that axis to the value you have just found using the appropriate command.



Z axis:

To calibrate the z axis hone the bit to the height of the cutting surface using a piece of paper as you would with a regular piece of material. Next get a piece of material that is perfectly flat with the same height throughout the entire piece (metal is recommended). Measure the height of this piece and place on piece of paper on cutting surface. Raise the bit the height of the piece of material you are using by entering the height as the step size and raising the bit that amount. Measure the distance traveled (by measuring the difference from top of piece to bit and add or subtract to height of bit depending on whether higher or lower) and divide the height of the piece of material by distance traveled. Multiply the quotient by the value for the steps/mm on the Z axis and enter this value as the steps/mm for the Z axis using the appropriate command.



Testing:

It is good to test all of these axes by measuring the distance the bit moves when you enter a specific distance. It may also be good to repeat the calibration process for all axes to get the machine to be very precise. When you are finished click close where open was before to close the connection, the settings you entered will be saved.

Interfacing with Grbl

Edit New Page

Sonny Jeon edited this page on Jan 21 · 27 revisions

Grbl v1.1 has been released at our new site! The old site will eventually be phased out.

Quick-Links

- Start Up Message
- Grbl `$` Help Message
- Grbl Response Meanings
- Other Grbl Messages
- Writing an Interface for Grbl
- Streaming a G-Code Program to Grbl
- Interacting with Grbl's Systems

#Grbl Interface

The interface for Grbl is fairly simple and straightforward. We have taken steps to try to make it as easy as possible for new users to get started, and for GUI developers to write their own custom interfaces to Grbl.

Everything communicates through the serial interface on the Arduino USB port. You just need to connect your Arduino to your computer with a USB cable. Use any standard serial terminal program to connect to Grbl, such as: the Arduino IDE serial monitor, Coolterm, puTTY, etc. Or use one of the many great Grbl GUIs out there in the Internet wild.

Just about every user interaction with Grbl is performed by sending it a string of characters, followed by an enter. Grbl will then process the string, execute it accordingly, and then reply back with a response to tell you how it went. These strings include sending Grbl: a G-code block to execute, commands to configure Grbl's settings, to view how Grbl is doing, etc. At times, Grbl may not respond immediately. This happens only when Grbl is busy doing something else, or waiting for some room to clear in its look-ahead planner buffer so it can finish processing the previous line sent.

However, one exception to this are Grbl's real-time commands. These are picked directly from the incoming serial stream to execute immediately and asynchronously. See our Configuring Grbl wiki to see what they are and how they work.

Start Up Message

Clone this wiki locally

`https://github.com/grbl/g`

Clone in Desktop

```
Grbl vX.Xx ['$' for help]
```

The start up message always prints upon startup, after a reset, or at program end. Whenever you see this message, this also means that Grbl has completed re-initializing all systems, so everything starts out the same every time you use Grbl.

- `vX.Xx` indicates the major version number, followed by a minor version letter. The major version number indicates the general release, while the letter simply indicates a feature update or addition from the preceding minor version letter.
- Bug fix revisions are tracked by the build info number, printed when an `$I` command is sent. These revisions don't update the version number and are given by date revised in year, month, and day, like so `20140820`.

Grbl `$` Help Message

Every string Grbl receives is assumed to be a G-code block/line for it to execute, except for some special system commands Grbl uses for configuration, provide feedback to the user on what and how it's doing, or perform some task such as a homing cycle. To see a list of these system commands, type `$` followed by an enter, and Grbl will respond with:

```
$$ (view Grbl settings)
$# (view # parameters)
$G (view parser state)
$I (view build info)
$N (view startup blocks)
$x=value (save Grbl setting)
$Nx=line (save startup block)
$C (check gcode mode)
$X (kill alarm lock)
$H (run homing cycle)
~ (cycle start)
! (feed hold)
? (current status)
ctrl-x (reset Grbl)
```

- Check out our Configuring Grbl wiki page to find out what all of these commands mean and how to use them.

#Grbl Response Meanings

Every G-code block sent to Grbl and Grbl system commands (excluding the real-time commands) will respond with how it went. This section will describe Grbl's responses and their meanings.

- **`ok`** : All is good! Everything in the last line was understood by Grbl and was successfully processed and executed.

- **`error:Expected command letter`** : G-code is composed of G-code "words", which consists of a letter followed by a number value. This error occurs when the letter prefix of a G-code word is missing in the G-code block (aka line).

- **`error:Bad number format`** : The number value suffix of a G-code word is missing in the G-code block, or when configuring a `$Nx=line` or `$x=val` Grbl setting and the `x` is not a number value.

- **`error:Invalid statement`** : The issued Grbl `$` system command is not recognized or is invalid.

- **`error:Value < 0`** : The value of a `$x=val` Grbl setting, `F` feed rate, `N` line number, `P` word, `T` tool number, or `S` spindle speed is negative.
- **`error:Setting disabled`** : Homing is disabled when issuing a `$H` command.
- **`error:Value < 3 usec`** : Step pulse time length cannot be less than 3 microseconds (for technical reasons).
- **`error:EEPROM read fail. Using defaults`** : If Grbl can't read data contained in the EEPROM, this error is returned. Grbl will also clear and restore the effected data back to defaults.
- **`error:Not idle`** : Certain Grbl `$` commands are blocked depending Grbl's current state, or what its doing. In general, Grbl blocks any command that fetches from or writes to the EEPROM since the AVR microcontroller will shutdown all of the interrupts for a few clock cycles when this happens. There is no work around, other than blocking it. This ensures both the serial and step generator interrupts are working smoothly throughout operation.
- **`error:Alarm lock`** : Grbl enters an `ALARM` state when Grbl doesn't know where it is and will then block all G-code commands from being executed. This error occurs if G-code commands are sent while in the alarm state. Grbl has two alarm scenarios: When homing is enabled, Grbl automatically goes into an alarm state to remind the user to home before doing anything; When something has went critically wrong, usually when Grbl can't guarantee positioning. This typically happens when something causes Grbl to force an immediate stop while its moving from a hard limit being triggered or a user commands an ill-timed reset.
- **`error:Homing not enabled`** : Soft limits cannot be enabled if homing is not enabled, because Grbl has no idea where it is when you startup your machine unless you perform a homing cycle.
- **`error:Line overflow`** : Grbl has to do everything it does within 2KB of RAM. Not much at all. So, we had to make some decisions on what's important. Grbl limits the number of characters in each line to less than 80 characters (70 in v0.8, 50 in v0.7 or earlier), excluding spaces or comments. The G-code standard mandates 256 characters, but Grbl simply doesn't have the RAM to spare. However, we don't think there will be any problems with this with all of the expected G-code commands sent to Grbl. This error almost always occurs when a user or CAM-generated G-code program sends position values that are in double precision (i.e. -2.003928578394852), which is not realistic or physically possible. Users and GUIs need to send Grbl floating point values in single precision (i.e. -2.003929) to avoid this error.
- **`error:Modal group violation`** : The G-code parser has detected two G-code commands that belong to the same modal group in the block/line. Modal groups are sets of G-code commands that mutually exclusive. For example, you can't issue both a `G0` rapids and `G2` arc in the same line, since they both need to use the XYZ target position values in the line. LinuxCNC.org has some great documentation on modal groups.
- **`error:Unsupported command`** : The G-code parser doesn't recognize or support one of the G-code commands in the line. Check your G-code program for any unsupported commands and either remove them or update them to be compatible with Grbl.
- **`error:Undefined feed rate`** : There is no feed rate programmed, and a G-code command that requires one is in the block/line. The G-code standard mandates `F` feed rates to be undefined upon a reset or when switching from inverse time mode to units mode. Older Grbl versions had a default feed rate setting, which was illegal and was removed in Grbl v0.9.

- **`error:Invalid gcode ID:XX`** : To save some flash space, Grbl v0.9 installed some cryptic invalid G-code numbers to indicate uncommon G-code programming errors. Storing full strings to describe all of the errors would use up the rest of the precious flash space we have to work with. The most common G-code errors, listed above, are still printed in human-readable strings though.

ID	Description
23	A `G` or `M` command value in the block is not an integer. For example, `G4` can't be `G4.13`. Some G-code commands are floating point (`G92.1`), but these are ignored.
24	Two G-code commands that both require the use of the `XYZ` axis words were detected in the block.
25	A G-code word was repeated in the block.
26	A G-code command implicitly or explicitly requires `XYZ` axis words in the block, but none were detected.
27	The G-code protocol mandates `N` line numbers to be within the range of 1-99,999. We think that's a bit silly and arbitrary. So, we increased the max number to 9,999,999. This error occurs when you send a number more than this.
28	A G-code command was sent, but is missing some important `P` or `L` value words in the line. Without them, the command can't be executed. Check your G-code.
29	Grbl supports six work coordinate systems `G54-G59`. This error happens when trying to use or configure an unsupported work coordinate system, such as `G59.1`, `G59.2`, and `G59.3`.
30	The `G53` G-code command requires either a `G0` seek or `G1` feed motion mode to be active. A different motion was active.
31	There are unused axis words in the block and `G80` motion mode cancel is active.
32	A `G2` or `G3` arc was commanded but there are no `XYZ` axis words in the selected plane to trace the arc.
33	The motion command has an invalid target. `G2`, `G3`, and `G38.2` generates this error. For both probing and arcs traced with the radius definition, the current position cannot be the same as the target. This also errors when the arc is mathematically impossible to trace, where the current position, the target position, and the radius of the arc doesn't define a valid arc.
34	A `G2` or `G3` arc, traced with the radius definition, had a mathematical error when computing the arc geometry. Try either breaking up the arc into semi-circles or quadrants, or redefine them with the arc offset definition.
35	A `G2` or `G3` arc, traced with the offset definition, is missing the `IJK` offset word in the selected plane to trace the arc.
36	There are unused, leftover G-code words that aren't used by any command in the block.
37	The `G43.1` dynamic tool length offset command cannot apply an offset to an axis other than its configured axis. The Grbl default axis is the Z-axis.

Other Grbl Messages

Along with the normal responses from user input, Grbl provides additional messages for important feedback of its current state. These messages are organized into three general classes: ALARM messages, feedback messages, and real-time status messages.

Alarms

Alarm is an emergency state. Something has gone terribly wrong when these occur. Typically, they are caused by limit error when the machine has moved or wants to move outside the machine space and crash into something. They also report problems if Grbl is lost and can't guarantee positioning or a probe command has failed. Once in alarm-mode, Grbl will lock out and shut down everything until the user issues a reset. Even after a reset, Grbl will remain in alarm-mode, block all G-code from being executed, but allows the user to override the alarm manually. This is to ensure the user knows and acknowledges the problem and has taken steps to fix or account for it.

All alarm messages start with **`ALARM:``**, followed by a brief description of the alarm cause.

- **`ALARM:Hard/soft limit`** : Hard and/or soft limits must be enabled for this error to occur. With hard limits, Grbl will enter alarm mode when a hard limit switch has been triggered and force kills all motion. Machine position will be lost and require re-homing. With soft limits, the alarm occurs when Grbl detects a programmed motion trying to move outside of the machine space, set by homing and the max travel settings. However, upon the alarm, a soft limit violation will instruct a feed hold and wait until the machine has stopped before issuing the alarm. Soft limits do not lose machine position because of this.

- **`ALARM:Abort during cycle`** : This alarm occurs when a user issues a soft-reset while the machine is in a cycle and moving. The soft-reset will kill all current motion, and, much like the hard limit alarm, the uncontrolled stop causes Grbl to lose position.

- **`ALARM:Probe fail`** : The `G38.2` straight probe command requires an alarm or error when the probe fails to trigger within the programmed probe distance. Grbl enters the alarm state to indicate to the user the probe has failed, but will not lose machine position, since the probe motion comes to a controlled stop before the error.

Feedback Messages

Feedback messages provide non-critical information on what Grbl is doing and/or what it needs. Not too complicated. Feedback message are always enclosed in `[]` brackets.

- **`[Reset to continue]`** : Sent after an alarm message to tell the user to reset Grbl as an acknowledgement that an alarm has happened.

- **`['$H'|'$X' to unlock]`** : After an alarm and the user has sent a reset, this feedback message is sent after the startup message to tell the user that all G-code commands are locked out, until the user unlocks it manually with the `$X` command or performs a homing cycle. Also, if a user has homing enabled, this message also is sent upon a fresh power-up to indicate the user needs to home the machine before doing anything else.

- **`[Caution: Unlocked]`** : The alarm mode can be manually over-ridden by the user issuing a `$X` command. This feedback message is sent when the user overrides the alarm.

- **`[Enabled]`** : A simple feedback message to indicate to the user that a Grbl state or mode has been enabled.

- **`[Disabled]`** : Same as above, but notifies state or mode has been disabled.

Other Messages:

- **`[PRB:0.000,0.000,1.492:1]`** : This is a little out of place, but as a service to GUIs, Grbl will immediately send a feedback message containing the triggered probe position upon a successful `G38.2` straight probe command. This data can also be viewed in the parameters print out called by `$#`.

- *(Grbl v0.9i or later)* The `:1` suffix value is a boolean that denotes whether the last probe cycle was successful or not.

- **`[G0 G54 G17 G21 G90 G94 M0 M5 M9 T0 F0. S0.]`** : When a `$G` system command is sent, Grbl replies with a message containing the current G-code parser modal state.

- *(Grbl 0.9i and later)* `S0.` denotes spindle speed and prints only if variable spindle speed is enabled.

- **`$# View Parameters`** : When a `$#` system command is sent, Grbl replies with several messages containing the current G-code parameter, which includes the work coordinate offsets, pre-defined positions, G92 coordinate offset, tool length offset, and last probe position. All of this data is printed with the feedback message `[]` brackets.

Writing an Interface for Grbl

FOR DEVELOPERS ONLY: This section outlines the recommended ways to setup a communications and streaming protocol with Grbl for a GUI.

The general interface for Grbl has been described above, but what's missing is how to run an entire G-code program on Grbl, when it doesn't seem to have an upload feature. Or, how to build a decent GUI with real-time feedback. This is where this section fits in. Early on, users fiercely requested for flash drive, external RAM, LCD support, joysticks, or network support so they can upload a g-code program and run it directly on Grbl. The general answer to that is, good ideas, but Grbl doesn't need them. Grbl already has nearly all of the tools and features to reliably communicate with a simple graphical user interface (GUI). Plus, we want to minimize as much as we can on what Grbl should be doing, because, in the end, Grbl needs to be concentrating on producing clean, reliable motion. That's it.

Streaming a G-Code Program to Grbl

Here we will describe three different streaming methods for Grbl GUIs, but really there's only two that we recommend using. One of the main problems with streaming to Grbl is the USB port itself. Arduinos and most all micro controllers use a USB-to-serial converter chip that, at times, behaves strangely and not typically how you'd expect, like USB packet buffering and delays that can wreak havoc to a streaming protocol. Another problem is how to deal with some of the latency and oddities of the PCs themselves, because none of them are truly real-time and always create micro-delays when executing other tasks. Regardless, we've come up with ways to ensure the G-code stream is reliable and simple.

Streaming Protocol: Simple Send-Response *[Recommended for Grbl v0.9+]*

The send-response streaming protocol is the most fool-proof and simplest method to stream a G-code program to Grbl. The host PC interface simply sends a line of G-code to Grbl and waits for an `ok` or `error:` response before sending the next line of G-code. So, no matter if Grbl needs to wait for room in the look-ahead planner buffer to finish parsing and executing the last line of G-code or if the the host computer is busy doing something, this guarantees both to the host PC and Grbl, the programmed G-code has been sent and received properly. An example of this protocol is published in our `simple_stream.py` script in our repository.

However, it's also the slowest of three outlined streaming protocols. Grbl essentially has two buffers between the execution of steps and the host PC interface. One of them is the serial receive buffer. This briefly stores up to 127 characters of data received from the host PC until Grbl has time to fetch and parse the line of G-code. The other buffer is the look-ahead planner buffer. This buffer stores up to 17 line motions that are acceleration-planned and optimized for step execution. Since the send-response protocol receives a line of G-code while the host PC waits for a response, Grbl's serial receive buffer is usually empty and under-utilized. If Grbl is actively running and executing steps, Grbl will immediately begin to execute and empty the look-ahead planner buffer, while it sends the response to the host PC, waits for the next line from the host PC, upon receiving it, parse and plan it, and add it to the end of the look-ahead buffer.

Although this communication lag may take only a fraction of a second, there is a cumulative effect, because there is a lag with every G-code block sent to Grbl. In certain scenarios, like a G-code program containing lots of sequential, very short, line segments with high feed rates, the cumulative lag can be large enough to empty and starve the look-ahead planner buffer within this time. This could lead to start-stop motion when the streaming can't keep up with G-code program execution. Also, since Grbl can only plan and optimize what's in the look-ahead planner buffer, the performance through these types of motions will never be full-speed, because look-ahead buffer will always be partially full when using this streaming method. If your expected application doesn't contain a lot of these short line segments with high feed rates, this streaming protocol should be more than adequate for a vast majority of applications, is very robust, and is a quick way to get started. However, we do not recommend using this method for Grbl versions v0.8 or prior due to some performance issues with these versions.

Streaming Protocol: Via Flow Control (XON/XOFF)

To avoid the risk of starving the look-ahead planner buffer, a flow control streaming protocol can be used to try to keep Grbl's serial receive buffer full, so that Grbl has immediate access to the next g-code line to parse and plan without having to wait for the host PC to send it. Flow control, also known as XON/XOFF software flow control, uses two special characters to tell the host PC when it has or doesn't have room in the serial receive buffer to receive more data. When there is room, usually at 20% full, the special character is sent to the host PC indicating ready-to-receive. The host PC will begin to send data until it receives the other stop-receive special character, usually at 80% full. Grbl's XON/XOFF software flow control feature may be enabled through the config.h, but is not officially supported for the following reasons.

While sound in logic, software flow control has a number of problems. The timing between Grbl and the host PC is almost never perfectly in sync, in large part due to the USB protocol and the USB-serial converter chips on every Arduino. This poses a big problem when sending and receiving these special flow-control characters. When Grbl's serial receive buffer is low, the time between when it sends the ready-to-receive character and when the host PC sends more data all depends everything in between. If the host PC is busy or the Arduino USB-serial converter is not sending the character on time, this lag can cause Grbl to wait for more serial data to come in before parsing and executing the next line of G-code. Even worse though, if the serial receive buffer is nearing full and the stop-receive character is sent, the host PC may not receive the signal in time to stop the data transfer and over-flow Grbl's serial buffer. This is bad and will corrupt the data stream.

Because the software flow-control method is dependent on the performance of the USB-serial converter on the Arduino and the host PC, the low and high watermarks for the ready-to-receive and stop-receive characters must be tuned for each case. Thus, it's not really a robust solution. In our experience with XON/XOFF software flow control, it absolutely **DOES NOT** work with Arduinos with the Atmega8U/16U USB-serial converter chips (on all current Arduinos from the Uno to Mega2560). For some reason, there are USB packet delays that are out of Grbl's control and almost always led to data corruption. However, XON/XOFF worked, but only on older

Arduinos or micro controllers that featured an FTDI RS232 USB-serial converter chip, such as the Duemilanove or controllers with an FTDI break-out board. The FTDI's firmware reliably sent the XON/XOFF special characters in time and on time. We're not sure why there is such a difference between them.

If you decide to use XON/XOFF software flow control for your GUI, keep in mind that, at the moment, it'll only really works with FTDI USB-serial converters. But, the great thing about this method is that you can connect with Grbl over a serial emulator program like Coolterm, enable XON/XOFF flow control, cut-and-paste an entire g-code program into it, and Grbl will execute it completely. (Nice but not really necessary.)

Streaming Protocol: Character-Counting *[Recommended with Reservations]*

To get the best of both worlds, the simplicity and reliability of the send-response method and assurance of maximum performance with software flow control, we came up with a simple character-counting protocol for streaming a G-code program to Grbl. It works like the send-response method, where the host PC sends a line of G-code for Grbl to execute and waits for a response, but, rather than needing special XON/XOFF characters for flow control, this protocol simply uses Grbl's responses as a way to reliably track how much room there is in Grbl's serial receive buffer. An example of this protocol is outlined in the `stream.py` streaming script in our repo.

The main difference between this protocol and the others is the host PC needs to maintain a standing count of how many characters it has sent to Grbl and then subtract the number of characters corresponding to the line executed with each Grbl response. Suppose there is a short G-code program that has 5 lines with 25, 40, 31, 58, and 20 characters (counting the line feed and carriage return characters too). We know Grbl has a 127 character serial receive buffer, and the host PC can send up to 127 characters without overflowing the buffer. If we let the host PC send as many complete lines as we can without over flowing Grbl's serial receive buffer, the first three lines of 25, 40, and 31 characters can be sent for a total of 96 characters. When Grbl responds, we know the first line has been processed and is no longer in the serial read buffer. As it stands, the serial read buffer now has the 40 and 31 character lines in it for a total of 71 characters. The host PC needs to then determine if it's safe to send the next line without overflowing the buffer. With the next line at 58 characters and the serial buffer at 71 for a total of 129 characters, the host PC will need to wait until more room has cleared from the serial buffer. When the next Grbl response comes in, the second line has been processed and only the third 31 character line remains in the serial buffer. At this point, it's safe to send the remaining last two 58 and 20 character lines of the g-code program for a total of 109.

While seemingly complicated, this character-counting streaming protocol is extremely effective in practice. It always ensures Grbl's serial read buffer is filled, while never overflowing it. It maximizes Grbl's performance by keeping the look-ahead planner buffer full by better utilizing the bi-directional data flow of the serial port, and it's fairly simple to implement as our `stream.py` script illustrates. We have stress-tested this character-counting protocol to extremes and it has not yet failed. Seemingly, only the speed of the serial connection is the limit.

UPDATE: Up until recently, we've recommended that Grbl GUIs use this streaming protocol. It is indeed very efficient and effective, but there are a couple of additional things interface writers should aware of. These are issues being worked on for the v1.0 release.

- *Since the GUI is preloading Grbl's serial RX buffer with commands, Grbl will continually execute all of the queued g-code in the RX serial buffer. The first problem is if there is an error at the beginning of the RX buffer, Grbl will continue to execute the remaining buffered g-code and the GUI won't be able to control what happens. The interim solution is to check all of the g-code via the $C check mode, so all errors are vetted prior to streaming.*
- *When Grbl stores data to EEPROM, the AVR requires all interrupts to be disabled during this*

> *write process, including the serial RX ISR. This means that if a g-code or Grbl `$` command writes to EEPROM, the data sent during the write may be lost. This is usually rare and typically occurs when streaming a G10 command inappropriately inside a program. For robustness, GUIs should track and detect these EEPROM write commands and handle them appropriately by waiting for the queue to finish executing before sending more data. Note that the simple send-response protocol doesn't not suffer from this issue.*

Interacting with Grbl's Systems

Along with streaming a G-code program, there a few more things to consider when writing a GUI for Grbl, such as how to use status reporting, real-time control commands, dealing with EEPROM, and general message handling.

Status Reporting

When a `?` character is sent to Grbl (no additional line feed or carriage return character required), it will immediately respond with something like `<Idle,MPos:0.000,0.000,0.000,WPos:0.000,0.000,0.000>` to report its state and current position. The `?` is always picked-off and removed from the serial receive buffer whenever Grbl detects one. So, these can be sent at any time. Also, to make it a little easier for GUIs to pick up on status reports, they are always encased by `<>` chevrons.

Developers can use this data to provide an on-screen position digital-read-out (DRO) for the user and/or to show the user a 3D position in a virtual workspace. We recommend querying Grbl for a `?` real-time status report at no more than 5Hz. 10Hz may be possible, but at some point, there are diminishing returns and you are taxing Grbl's CPU more by asking it to generate and send a lot of position data.

Grbl's status report is fairly simply in organization. It always starts with a word describing the machine state like `IDLE` (descriptions of these are available elsewhere in the Wiki). The following data values are usually in the order listed below and separated by commas, but may not be in the exact order or printed at all. Report output depends on the user's `$10` status report mask setting.

- `MPos:0.000,0.000,0.000` : Machine position listed as `X,Y,Z` coordinates. Units (mm or inch) depends on `$13` Grbl unit reporting setting.
- `WPos:0.000,0.000,0.000` : Work position listed as `X,Y,Z` coordinates. Units (mm or inch) depends on `$13` Grbl unit reporting setting.
- `Buf:0` : Number of motions queued in Grbl's planner buffer.
- `RX:0` : Number of characters queued in Grbl's serial RX receive buffer.

Real-Time Control Commands

The real-time control commands, `~` cycle start/resume, `!` feed hold, and `^X` soft-reset, all immediately signal Grbl to change its running state. Just like `?` status reports, these control characters are picked-off and removed from the serial buffer when they are detected and do not require an additional line-feed or carriage-return character to operate.

EEPROM Issues

EEPROM access on the Arduino AVR CPUs turns off all of the interrupts while the CPU *writes* to EEPROM. This poses a problem for certain features in Grbl, particularly if a user is streaming and running a g-code program, since it can pause the main step generator interrupt from executing on time. Most of the EEPROM access is restricted by Grbl when it's in certain states, but there are some things that developers need to know.

- Settings should not be streamed with the character-counting streaming protocols. Only the simple send-response protocol works. This is because during the EEPROM write, the AVR CPU also shuts-down the serial RX interrupt, which means data can get corrupted or lost. This is safe with the send-response protocol, because it's not sending data after commanding Grbl to save data.

For reference:

- Grbl's EEPROM write commands: `G10 L2` , `G10 L20` , `G28.1` , `G30.1` , `$x=` , `$I=` , `$Nx=` , `$RST=`
- Grbl's EEPROM read commands: `G54-G59` , `G28` , `G30` , `$$` , `$I` , `$N` , `$#`

Message Handling

Most of the feedback from Grbl fits into nice categories so GUIs can easily tell what is what. Here's how they are organized:

- `ok` : Standard all-is-good response to a single line sent to Grbl.
- `error:` : Standard error response to a single line sent to Grbl.
- `ALARM:` : A critical error message that occurred. All processes stopped until user acknowledgment.
- `[]` : All feedback messages are sent in brackets. These include parameter and g-code parser state print-outs.
- `<>` : Status reports are sent in chevrons.

There are few things that don't fit neatly into this setup at the moment. In the next version, we'll try to make this more universal, but for now, your GUIs will need to manually account for these:

- The startup message.
- `$` Help print-out.
- `$N` Start-up blocks execution after the startup message.
- The `$$` view Grbl settings print-out.

G-code Error Handling

As of Grbl v0.9, the g-code parser is fully standards-compilant with complete error-checking. When a G-code parser detects an error in a G-code block/line, the parser will dump everything in the block from memory and report an `error:` back to the user or GUI. This dump can pose problematic, because the bad G-code block may have contained some valuable positioning commands or feed rate settings.

It's highly recommended to do what all professional CNC controllers do when they detect an error in the G-code program, ***halt***. Don't do anything further until the user has modified the G-code and fixed the error in their program. Otherwise, bad things could happen.

As a service to GUIs, Grbl has a "check G-code" mode, enabled by the `$C` system command. GUIs can stream a G-code program to Grbl, where it will parse it, error-check it, and report `ok` 's and `errors:` 's without powering on anything or moving. So GUIs can pre-check the programs before streaming them for real. To disable the "check G-code" mode, send another `$C` system command and Grbl will automatically soft-reset to flush and re-initialize the G-code parser and the rest of the system. This perhaps should be run in the background when a user first loads a program, before a user sets up his machine. This flushing and re-initialization clears `G92` 's by G-code standard, which some users still incorrectly use to set their part zero.

Jogging

Unfortunately, Grbl doesn't have a proper jogging interface, at least for now. This was to conserve precious flash space for the development of Grbl v0.9, but may be installed in the next release of Grbl. However, authors of Grbl GUIs have come up with ways to simulate jogging with Grbl by sending incremental motions, such as `G91 X0.1`, with each jogging click or key press. This works pretty well, but, if uncontrolled, a user can easily queue up more motions than they want without realizing it and move well-past their desired location.

The general methodology that has been shown to work is to simply restrict the number of jogging commands sent to Grbl. This can be done by disabling key repeating when held down. The planner buffer queue size can be tracked such that only a handful of motions can be queued and executed.

Synchronization

For situations when a GUI needs to run a special set of commands for tool changes, auto-leveling, etc, there often needs to be a way to know when Grbl has completed a task and the planner buffer is empty. The absolute simplest way to do this is to insert a `G4 P0.01` dwell command, where P is in seconds and must be greater than 0.0. This acts as a quick force-synchronization and ensures the planner buffer is completely empty before the GUI sends the next task to execute.

Source: https://github.com/grbl/grbl/wiki/Interfacing-with-Grbl

Grbl v1.1 Configuration

Sonny Jeon edited this page on Jun 2 · 14 revisions

Quick-Links:

- Grbl's Settings and What They Mean
- Quick Guide to Setting Up Your Machine for the First Time

Getting Started

First, connect to Grbl using the serial terminal of your choice.

Set the baud rate to **115200** as 8-N-1 (8-bits, no parity, and 1-stop bit.)

Once connected you should get the Grbl-prompt, which looks like this:

```
Grbl 1.1f ['$' for help]
```

Type $ and press enter to have Grbl print a help message. You should not see any local echo of the $ and enter. Grbl should respond with:

```
[HLP:$$ $# $G $I $N $x=val $Nx=line $J=line $SLP $C $X $H ~ ! ? ctrl-x]
```

The '$'-commands are Grbl system commands used to tweak the settings, view or change Grbl's states and running modes, and start a homing cycle. The last four **non**-'$' commands are realtime control commands that can be sent at anytime, no matter what Grbl is doing. These either immediately change Grbl's running behavior or immediately print a report of the important realtime data like current position (aka DRO).

Grbl Settings

$$ - View Grbl settings

To view the settings, type `$$` and press enter after connecting to Grbl. Grbl should respond with a list of the current system settings, as shown in the example below. All of these settings are persistent and kept in EEPROM, so if you power down, these will be loaded back up the next time you power up your Arduino.

The `x` of `$x=val` indicates a particular setting, while `val` is the setting value. In prior versions of Grbl, each setting had a description next to it in `()` parentheses, but Grbl v1.1+ no longer includes them unfortunately. This was done to free up precious flash memory to add the new features available in v1.1. However, most good GUIs will help out by attaching descriptions for you, so you know what you are looking at.

```
$0=10
$1=25
$2=0
$3=0
$4=0
$5=0
$6=0
$10=1
$11=0.010
$12=0.002
$13=0
$20=0
$21=0
$22=1
$23=0
$24=25.000
$25=500.000
$26=250
$27=1.000
$30=1000.
$31=0.
$32=0
$100=250.000
$101=250.000
$102=250.000
$110=500.000
$111=500.000
$112=500.000
$120=10.000
$121=10.000
$122=10.000
$130=200.000
$131=200.000
$132=200.000
```

$x=val - Save Grbl setting

The `$x=val` command saves or alters a Grbl setting, which can be done manually by sending this command when connected to Grbl through a serial terminal program, but most Grbl GUIs will do this for you as a user-friendly feature.

To manually change e.g. the microseconds step pulse option to 10us you would type this, followed by an enter:

```
$0=10
```

If everything went well, Grbl will respond with an 'ok' and this setting is stored in EEPROM and will be retained forever or until you change them. You can check if Grbl has received and stored your setting correctly by typing `$$` to view the system settings again.

Grbl's `$x=val` settings and what they mean

NOTE: From Grbl v0.9 to Grbl v1.1, only `$10` status reports changed and new `$30` / `$31` spindle rpm max/min and `$32` laser mode settings were added. Everything else is the same.

$0 – Step pulse, microseconds

Stepper drivers are rated for a certain minimum step pulse length. Check the data sheet or just try some numbers. You want the shortest pulses the stepper drivers can reliably recognize. If the pulses are too long, you might run into trouble when running the system at very high feed and pulse rates, because the step pulses can begin to overlap each other. We recommend something around 10 microseconds, which is the default value.

$1 - Step idle delay, milliseconds

Every time your steppers complete a motion and come to a stop, Grbl will delay disabling the steppers by this value. **OR**, you can always keep your axes enabled (powered so as to hold position) by setting this value to the maximum 255 milliseconds. Again, just to repeat, you can keep all axes always enabled by setting `$1=255` .

The stepper idle lock time is the time length Grbl will keep the steppers locked before disabling. Depending on the system, you can set this to zero and disable it. On others, you may need 25-50 milliseconds to make sure your axes come to a complete stop before disabling. This is to help account for machine motors that do not like to be left on for long periods of time without doing something. Also, keep in mind that some stepper drivers don't remember which micro step they stopped on, so when you re-enable, you may witness some 'lost' steps due to this. In this case, just keep your steppers enabled via `$1=255` .

$2 – Step port invert, mask

This setting inverts the step pulse signal. By default, a step signal starts at normal-low and goes high upon a step pulse event. After a step pulse time set by `$0` , the pin resets to low, until the next step pulse event. When inverted, the step pulse behavior switches from normal-high, to low during the pulse, and back to high. Most users will not need to use this setting, but this can be useful for certain CNC-stepper drivers that have peculiar requirements. For example, an artificial delay between the direction pin and step pulse can be created by inverting the step pin.

This invert mask setting is a value which stores the axes to invert as bit flags. You really don't need to completely understand how it works. You simply need to enter the settings value for the axes you want to invert. For example, if you want to invert the X and Z axes, you'd send `$2=5` to Grbl and the setting should now read `$2=5 (step port invert mask:00000101)` .

Setting Value	Mask	Invert X	Invert Y	Invert Z
0	00000000	N	N	N
1	00000001	Y	N	N
2	00000010	N	Y	N
3	00000011	Y	Y	N
4	00000100	N	N	Y
5	00000101	Y	N	Y
6	00000110	N	Y	Y
7	00000111	Y	Y	Y

$3 – Direction port invert, mask

This setting inverts the direction signal for each axis. By default, Grbl assumes that the axes move in a positive direction when the direction pin signal is low, and a negative direction when the pin is high. Often, axes don't move this way with some machines. This setting will invert the direction pin signal for those axes that move the opposite way.

This invert mask setting works exactly like the step port invert mask and stores which axes to invert as bit flags. To configure this setting, you simply need to send the value for the axes you want to invert. Use the table above. For example, if want to invert the Y axis direction only, you'd send `$3=2` to Grbl and the setting should now read `$3=2 (dir port invert mask:00000010)`

$4 - Step enable invert, boolean

By default, the stepper enable pin is high to disable and low to enable. If your setup needs the opposite, just invert the stepper enable pin by typing `$4=1`. Disable with `$4=0`. (May need a power cycle to load the change.)

$5 - Limit pins invert, boolean

By default, the limit pins are held normally-high with the Arduino's internal pull-up resistor. When a limit pin is low, Grbl interprets this as triggered. For the opposite behavior, just invert the limit pins by typing `$5=1`. Disable with `$5=0`. You may need a power cycle to load the change.

NOTE: For more advanced usage, the internal pull-up resistor on the limit pins may be disabled in config.h.

$6 - Probe pin invert, boolean

By default, the probe pin is held normally-high with the Arduino's internal pull-up resistor. When the probe pin is low, Grbl interprets this as triggered. For the opposite behavior, just invert the probe pin by typing `$6=1`. Disable with `$6=0`. You may need a power cycle to load the change.

NOTE: If you invert your probe pin, you will need an external pull-down resistor wired in to the probe pin to prevent overloading it with current and frying it.

$10 - Status report, mask

This setting determines what Grbl real-time data it reports back to the user when a '?' status report is sent. This data includes current run state, real-time position, real-time feed rate, pin states, current override values, buffer states, and the g-code line number currently executing (if enabled through compile-time options).

By default, the new report implementation in Grbl v1.1+ will include just about everything in the standard status report. A lot of the data is hidden and will appear only if it changes. This increases efficiency dramatically over of the old report style and allows you to get faster updates and still get more data about your machine. The interface documentation outlines how it works and most of it applies only to GUI developers or the curious.

To keep things simple and consistent, Grbl v1.1 has only two reporting options. These are primarily here just for users and developers to help set things up.

- Position type may be specified to show either machine position (`MPos:`) or work position (`WPos:`), but no longer both at the same time. Enabling work position is useful in certain scenarios when Grbl is being directly interacted with through a serial terminal, but *machine position reporting should be used by default.*
- Usage data of Grbl's planner and serial RX buffers may be enabled. This shows the number of blocks or bytes available in the respective buffers. This is generally used to helps determine how Grbl is performing when testing out a streaming interface. *This should be disabled by default.*

Use the table below enables and disable reporting options. Simply add the values listed of what you'd like to enable, then save it by sending Grbl your setting value. For example, the default report with machine position and no buffer data reports setting is `$10=1`. If work position and buffer data are desired, the setting will be `$10=2` .

Report Type	Value	Description
Position Type	0	Enable `WPos:` Disable `MPos:` .
Position Type	1	Enable `MPos:` . Disable `WPos:` .
Buffer Data	2	Enabled `Buf:` field appears with planner and serial RX available buffer.

$11 - Junction deviation, mm

Junction deviation is used by the acceleration manager to determine how fast it can move through line segment junctions of a G-code program path. For example, if the G-code path has a sharp 10 degree turn coming up and the machine is moving at full speed, this setting helps determine how much the machine needs to slow down to safely go through the corner without losing steps.

How we calculate it is a bit complicated, but, in general, higher values gives faster motion through corners, while increasing the risk of losing steps and positioning. Lower values makes the acceleration manager more careful and will lead to careful and slower cornering. So if you run into problems where your machine tries to take a corner too fast, *decrease* this value to make it slow down when entering corners. If you want your machine to move faster through junctions, *increase* this value to speed it up. For curious people, hit this link to read about Grbl's cornering algorithm, which accounts for both velocity and junction angle with a very simple, efficient, and robust method.

$12 – Arc tolerance, mm

Grbl renders G2/G3 circles, arcs, and helices by subdividing them into teeny tiny lines, such that the arc tracing accuracy is never below this value. You will probably never need to adjust this setting, since `0.002mm` is well below the accuracy of most all CNC machines. But if you find that your circles are too crude or arc tracing is performing slowly, adjust this setting. Lower values give higher precision but may lead to performance issues by overloading Grbl with too many tiny lines. Alternately, higher values traces to a lower precision, but can speed up arc performance since Grbl has fewer lines to deal with.

For the curious, arc tolerance is defined as the maximum perpendicular distance from a line segment with its end points lying on the arc, aka a chord. With some basic geometry, we solve for the length of the line segments to trace the arc that satisfies this setting. Modeling arcs in this way is great, because the arc line segments automatically adjust and scale with length to ensure optimum arc tracing performance, while never losing accuracy.

$13 - Report inches, boolean

Grbl has a real-time positioning reporting feature to provide a user feedback on where the machine is exactly at that time, as well as, parameters for coordinate offsets and probing. By default, it is set to report in mm, but by sending a `$13=1` command, you send this boolean flag to true and these reporting features will now report in inches. `$13=0` to set back to mm.

$20 - Soft limits, boolean

Soft limits is a safety feature to help prevent your machine from traveling too far and beyond the limits of travel, crashing or breaking something expensive. It works by knowing the maximum travel limits for each axis and where Grbl is in machine coordinates. Whenever a new G-code motion is sent to Grbl, it checks whether or not you accidentally have exceeded your machine space. If you do, Grbl will issue an immediate feed hold wherever it is, shutdown the spindle and coolant, and then set the system alarm indicating the problem. Machine position will be retained afterwards, since it's not due to an immediate forced stop like hard limits.

NOTE: Soft limits requires homing to be enabled and accurate axis maximum travel settings, because Grbl needs to know where it is. `$20=1` to enable, and `$20=0` to disable.

$21 - Hard limits, boolean

Hard limit work basically the same as soft limits, but use physical switches instead. Basically you wire up some switches (mechanical, magnetic, or optical) near the end of travel of each axes, or where ever you feel that there might be trouble if your program moves too far to where it shouldn't. When the switch triggers, it will immediately halt all motion, shutdown the coolant and spindle (if connected), and go into alarm mode, which forces you to check your machine and reset everything.

To use hard limits with Grbl, the limit pins are held high with an internal pull-up resistor, so all you have to do is wire in a normally-open switch with the pin and ground and enable hard limits with `$21=1` . (Disable with `$21=0` .) We strongly advise taking electric interference prevention measures. If you want a limit for both ends of travel of one axes, just wire in two switches in parallel with the pin and ground, so if either one of them trips, it triggers the hard limit.

Keep in mind, that a hard limit event is considered to be critical event, where steppers immediately stop and will have likely have lost steps. Grbl doesn't have any feedback on position, so it can't guarantee it has any idea where it is. So, if a hard limit is triggered, Grbl will go into an infinite loop ALARM mode, giving you a chance to check your machine and forcing you to reset Grbl. Remember it's a purely a safety feature.

$22 - Homing cycle, boolean

Ahh, homing. For those just initiated into CNC, the homing cycle is used to accurately and precisely locate a known and consistent position on a machine every time you start up your Grbl between sessions. In other words, you know exactly where you are at any given time, every time. Say you start machining something or are about to start the next step in a job and the power goes out, you re-start Grbl and Grbl has no idea where it is due to steppers being open-loop control. You're left with the task of figuring out where you are. If you have homing, you always have the machine zero reference point to locate from, so all you have to do is run the homing cycle and resume where you left off.

To set up the homing cycle for Grbl, you need to have limit switches in a fixed position that won't get bumped or moved, or else your reference point gets messed up. Usually they are setup in the farthest point in +x, +y, +z of each axes. Wire your limit switches in with the limit pins, add a recommended RC-filter to help reduce electrical noise, and enable homing. If you're curious, you can use your limit switches for both hard limits AND homing. They play nice with each other.

Prior to trying the homing cycle for the first time, make sure you have setup everything correctly, otherwise homing may behave strangely. First, ensure your machine axes are moving in the correct directions per Cartesian coordinates (right-hand rule). If not, fix it with the `$3` direction invert setting. Second, ensure your limit switch pins are not showing as 'triggered' in Grbl's status reports. If are, check your wiring and settings. Finally, ensure your `$13x` max travel settings are somewhat accurate (within 20%), because Grbl uses these values to determine how far it should search for the homing switches.

By default, Grbl's homing cycle moves the Z-axis positive first to clear the workspace and then moves both the X and Y-axes at the same time in the positive direction. To set up how your homing cycle behaves, there are more Grbl settings down the page describing what they do (and compile-time options as well.)

Also, one more thing to note, when homing is enabled. Grbl will lock out all G-code commands until you perform a homing cycle. Meaning no axes motions, unless the lock is disabled ($X) but more on that later. Most, if not all CNC controllers, do something similar, as it is mostly a safety feature to prevent users from making a positioning mistake, which is very easy to do and be saddened when a mistake ruins a part. If you find this annoying or find any weird bugs, please let us know and we'll try to work on it so everyone is happy. :)

NOTE: Check out config.h for more homing options for advanced users. You can disable the homing lockout at startup, configure which axes move first during a homing cycle and in what order, and more.

$23 - Homing dir invert, mask

By default, Grbl assumes your homing limit switches are in the positive direction, first moving the z-axis positive, then the x-y axes positive before trying to precisely locate machine zero by going back and forth slowly around the switch. If your machine has a limit switch in the negative direction, the homing direction mask can invert the axes' direction. It works just like the step port invert and direction port invert masks, where all you have to do is send the value in the table to indicate what axes you want to invert and search for in the opposite direction.

$24 - Homing feed, mm/min

The homing cycle first searches for the limit switches at a higher seek rate, and after it finds them, it moves at a slower feed rate to home into the precise location of machine zero. Homing feed rate is that slower feed rate. Set this to whatever rate value that provides repeatable and precise machine zero locating.

$25 - Homing seek, mm/min

Homing seek rate is the homing cycle search rate, or the rate at which it first tries to find the limit switches. Adjust to whatever rate gets to the limit switches in a short enough time without crashing into your limit switches if they come in too fast.

$26 - Homing debounce, milliseconds

Whenever a switch triggers, some of them can have electrical/mechanical noise that actually 'bounce' the signal high and low for a few milliseconds before settling in. To solve this, you need to debounce the signal, either by hardware with some kind of signal conditioner or by software with a short delay to let the signal finish bouncing. Grbl performs a short delay, only homing when locating machine zero. Set this delay value to whatever your switch needs to get repeatable homing. In most cases, 5-25 milliseconds is fine.

$27 - Homing pull-off, mm

To play nice with the hard limits feature, where homing can share the same limit switches, the homing cycle will move off all of the limit switches by this pull-off travel after it completes. In other words, it helps to prevent accidental triggering of the hard limit after a homing cycle. Make sure this value is large enough to clear the limit switch. If not, Grbl will throw an alarm error for failing to clear it.

$30 - Max spindle speed, RPM

This sets the spindle speed for the maximum 5V PWM pin output. For example, if you want to set

10000rpm at 5V, program `$30=10000` . For 255rpm at 5V, program `$30=255` . If a program tries to set a higher spindle RPM greater than the `$30` max spindle speed, Grbl will just output the max 5V, since it can't go any faster. By default, Grbl linearly relates the max-min RPMs to 5V-0.02V PWM pin output in 255 equally spaced increments. When the PWM pin reads 0V, this indicates spindle disabled. Note that there are additional configuration options are available in config.h to tweak how this operates.

$31 - Min spindle speed, RPM

This sets the spindle speed for the minimum 0.02V PWM pin output (0V is disabled). Lower RPM values are accepted by Grbl but the PWM output will not go below 0.02V, except when RPM is zero. If zero, the spindle is disabled and PWM output is 0V.

$32 - Laser mode, boolean

When enabled, Grbl will move continuously through consecutive `G1` , `G2` , or `G3` motion commands when programmed with a `S` spindle speed (laser power). The spindle PWM pin will be updated instantaneously through each motion without stopping. Please read the GRBL laser documentation and your laser device documentation prior to using this mode. Lasers are very dangerous. They can instantly damage your vision permanantly and cause fires. Grbl does not assume any responsibility for any issues the firmware may cause, as defined by its GPL license.

When disabled, Grbl will operate as it always has, stopping motion with every `S` spindle speed command. This is the default operation of a milling machine to allow a pause to let the spindle change speeds.

$100, $101 and $102 – [X,Y,Z] steps/mm

Grbl needs to know how far each step will take the tool in reality. To calculate steps/mm for an axis of your machine you need to know:

- The mm traveled per revolution of your stepper motor. This is dependent on your belt drive gears or lead screw pitch.
- The full steps per revolution of your steppers (typically 200)
- The microsteps per step of your controller (typically 1, 2, 4, 8, or 16). *Tip: Using high microstep values (e.g., 16) can reduce your stepper motor torque, so use the lowest that gives you the desired axis resolution and comfortable running properties.*

The steps/mm can then be calculated like this: `steps_per_mm = (steps_per_revolution*microsteps)/mm_per_rev`

Compute this value for every axis and write these settings to Grbl.

$110, $111 and $112 – [X,Y,Z] Max rate, mm/min

This sets the maximum rate each axis can move. Whenever Grbl plans a move, it checks whether or not the move causes any one of these individual axes to exceed their max rate. If so, it'll slow down the motion to ensure none of the axes exceed their max rate limits. This means that each axis has its own independent speed, which is extremely useful for limiting the typically slower Z-axis.

The simplest way to determine these values is to test each axis one at a time by slowly increasing max rate settings and moving it. For example, to test the X-axis, send Grbl something like `G0 X50` with enough travel distance so that the axis accelerates to its max speed. You'll know you've hit the max rate threshold when your steppers stall. It'll make a bit of noise, but shouldn't hurt your motors. Enter a setting a 10-20% below this value, so you can account for wear, friction, and the mass of your workpiece/tool. Then, repeat for your other axes.

NOTE: This max rate setting also sets the G0 seek rates.

$120, $121, $122 – [X,Y,Z] Acceleration, mm/sec^2

This sets the axes acceleration parameters in mm/second/second. Simplistically, a lower value makes Grbl ease slower into motion, while a higher value yields tighter moves and reaches the desired feed rates much quicker. Much like the max rate setting, each axis has its own acceleration value and are independent of each other. This means that a multi-axis motion will only accelerate as quickly as the lowest contributing axis can.

Again, like the max rate setting, the simplest way to determine the values for this setting is to individually test each axis with slowly increasing values until the motor stalls. Then finalize your acceleration setting with a value 10-20% below this absolute max value. This should account for wear, friction, and mass inertia. We highly recommend that you dry test some G-code programs with your new settings before committing to them. Sometimes the loading on your machine is different when moving in all axes together.

$130, $131, $132 – [X,Y,Z] Max travel, mm

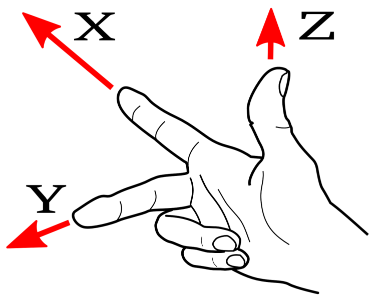
This sets the maximum travel from end to end for each axis in mm. This is only useful if you have soft limits (and homing) enabled, as this is only used by Grbl's soft limit feature to check if you have exceeded your machine limits with a motion command.

Quick Guide to Setting Up Your Machine for the First Time

Grbl's default configuration is intentionally very generic to help ensure users can see successful motion without having to tweak settings. Generally, the first thing you'll want to do is get your stepper motors running, usually without it connected to the CNC. Wire Grbl to your stepper drivers and stepper motors according to your manufacturer guidelines. Connect to Grbl through a serial terminal or one of many Grbl GUIs. Send some `G1` or `G0` commands to Grbl. You should see your stepper motor rotating. If you are having trouble with your stepper motors, try the following:

- Ensure everything is wired and powered correctly per your stepper driver manufacturer guidelines.
- If your steppers are mounted in your CNC already, ensure your axes move freely and don't obviously bind. If you can't easily tell, try removing your steppers and check if they run under no load.
- Ensure your stepper motors and axes linear mechanisms are all tight and secure. Small set screws on drivetrain components becoming loose is a very common problem. Re-tighten and try applying some non-permenant thread locker (Loctite blue) if it continually loosens.
- For more difficult issues, try the process of elimination to quickly isolate the problem. Start by disconnecting everything from the Arduino. Test if Grbl is operating ok by itself. Then, add one thing at a time and test.
- If your steppers are powered and making a grinding noise when trying to move, try lowering the '$' acceleration and max rate settings. This sound is a sign that your steppers is losing steps and not able to keep up due too much torque load or going too fast.
- Grbl's default step pulse settings cover the vast majority of stepper drivers on the market. While very uncommon, check these settings if you are still experiencing problems or have a unusual setup.

Next, you will need to make sure your machine is moving in the correct directions according to a Cartesian(XYZ) coordinate frame and satisfies the right-hand rule, as shown:



Mount your stepper motors into your CNC, if you haven't already done so. Send Grbl some motion commands, such as `G91 G0 X1` or `G91 G0 X-1` , which will move the x-axis +1mm and -1mm, respectively. Check all axes. If an axis is not moving correctly, alter the `$3` direction port mask setting to invert the direction.

If you are unfamiliar with how coordinate frames are setup on CNC machines, see this great diagram by LinuxCNC. Just keep in mind that motions are *relative* to the tool. So on a typical CNC gantry router, the tool will move rather than the fixed table. If the x-axis is aligned positive to the right, a positive motion command will move the tool to the right. Whereas, a moving table with a fixed tool will move the table to the left for the same command, because the tool is moving to the right relative to the table.

Finally, tune your settings to get close to your desired or max performance. Start by ensuring your `$100` , `$101` , and `$102` axes step/mm settings are correct for your setup. This is dependent on your stepper increments, micro steps on your driver, and mechanical parameters. There are multiple resources online to show you how to compute this for your particular machine, if your machine manufacturer has not supplied this for you. Tweak your `$11x` acceleration and `$12x` max rate settings to improve performance. Set to no greater than 80% of absolute max to account for inertia and cutting forces. Set your `$13x` max travel settings if you plan on using homing or soft limits. It's recommended to enter something approximately close to actual travel now to avoid problems in the future.

At this point, you're pretty much ready to get going! Grbl can now move your CNC machine and run g-code jobs. If you need to add more features, such as limit switches for homing or hard limits or spindle/laser control. There are other Wiki pages to help you that. Good luck and have fun!