Interfacing with Grbl

chamnit edited this page on 9 Sep · 19 revisions

Quick-Links

Grbl Interface

The interface for Grbl is fairly simple and straightforward. We taken steps to try to make it as easy as can 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, to configure Grbl's settings, 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

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

The start up message always prints upon startup, after a reset, or program end. Whenever you see this message, this also means that Grbl has completed re-initializing all systems, so everything starts out the same everytime 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 its doing, or perform some task like 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), 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 erros, 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

ID	Description
	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]` : 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 `$#` .
- `[G0 G54 G17 G21 G90 G94 M0 M5 M9 T0 F0.]` : When a `$G` system command is sent, Grbl replies with a message containing the current g-code parser modal state.
- `$# View Parameters` : When a `$#` 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 Call-Response *[Recommended for Grbl v0.9+]*

The call-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 repo.

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 call-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 most applications 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 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 only on older Arduinos or micro controllers that featured an FTDI RS232 USB-serial converter chip, like 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: Advanced Character-Counting *[Recommended!]*

To get the best of both worlds, the simplicity and reliability of the call-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 call-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 101.

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, 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. We recommend that all GUIs used this character-counting streaming protocol.

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.

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 reads and 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 can't be streamed to Grbl with either the XON/XOFF software flow control or advanced character-counting streaming protocols. Only the simple call-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. The call-response protocol doesn't send any data until a response comes back.

- When changing work coordinates or accessing the `G28` / `G30` predefined positions, Grbl has to fetch them from EEPROM. There is a small chance this access can pause the stepper or serial receive interrupt long enough to cause motion issues, but since it only fetches 12 bytes at a time at 2 cycles per fetch, the chances are very small that this will do anything to how Grbl runs. We just suggest keeping an eye on this and report to us any issues you might think are related to this.

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 is slated for the next release of Grbl. However, current 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 a user can easily queue up more motions than they want without realizing it and move well-past their desired location. To prevent this, GUIs should either prevent more than one (or a few) jogging motion(s) to be in queue before sending the next command or create a jog cancel feature, which will send Grbl a feed hold command and a soft-reset after the GUI has detected that the feed hold has completed.

The latter will flush the planner and maintain position, but isn't exactly clean and tidy. Whatever the jogging implementation or solution, just keep in mind this potential jogging queue issue.