# egi-pynetstation Documentation

Joshua B. Teves

January 23, 2023

## Contents

## 1 Introduction

This document is designed to guide you to some of the design decisions (and hardware quirks) of the EGI MRI-compatible EEG amplifier. General thoughts about how the codebase is designed and conceived can be found in this introductory section, with specifics on each module in the next section.

### 1.1 Package Purpose

The general purpose of the package is to allow a user to send commands to the amplifier through Python, and therefore PsychoPy (which this package is included in). There was originally a version of this code available for Python2, but not Python3. Now that most people use Python3, which is not backwards compatible with Python2, the original package was not working. Additionally, the old package did not support the Network Time Protocol, which allows high-precision clock synchronization. If you are not familiar with the idea, clocks drift in time from one another on most machines. This means that if your experiment depends on millisecond precision (which in EEG they often do), then you must account for this clock drift. The common solution is to use Network Time Protocol (NTP), and EGI provides a server to synchronize clocks with in the 400 Series amplifiers. It is worth noting that while this was not the initial intent of the package, pynetstation also became something of a documentation effort for how the amplifier processes events, as the SDK included in the repository is actually quite wrong. Differences between the SDK and what we found works in practice are documented in Section 2.2, as well as with code comments where applicable.

## 1.2 Differences between PyNetStation and PsychToolbox

There exists another implementation of a way of controlling the amplifier in Matlab-based Pscyhtoolbox (PTB). In the PTB implementation, the design is quite radically different. The PTB developers have had some sort of extensive exchange with Mark Moran at EGI, and ultimately concluded that they could get 1ms time resolution with their code. We have been unable to replicate this result. In order to achieve this, they create a linear model of clock draft and query the clock regularly to update the model. In the pynetstation implementation, we perform an NTP synchronization and do *not* model drift. Instead, we recommend that users regularly perform a clock synchronization through NTP. The PTB implementation offers a non-NTP solution to synchronize the clocks, but we do not. In practice we have found that non-NTP solutions are not very stable or well-resolved, and have therefore avoided creating any for users as they may be confused about why they are not achieving the temporal precision they expect. (Or worse, never checking the precision and assuming it is high). Another key difference is that because the PTB implementation is in Matlab, they avoid checking for NetStation error responses to save time. As discussed in Section 1.3, this was not viewed as an option, so pynetstation checks for error returns and will halt the program (and calling experiment) if it receives one. An additional consequence of pynetstation being Python-based is that installation outside of PsychoPy is quite straight forward; users can pip install it. Because PTB is in Matlab, there is some complexity with mex-files and path organization within PTB itself. It can prove quite treacherous to set up a way to compare them against each other. In general, you will need to make sure you have a working PTB installation and that you use the 'NTPSync' option when perform clock synchronization with PTB. The simplest test is a photodiode over a flashing screen, comparing what time you record sending an event vs. what the time is recorded as in a file.

## 1.3 General Philosophical and Design Considerations

PyNetStation took the following requirements as a given based on its intended use case:

1. Python-3 compatible.
2. Use a simple object to represent the connection.
3. Use simple methods to begin recording, end recording, and send events.
4. Produce sensible error messages for the user.
5. Achieve a consistent latency between event send and recorded time over the course of a 30-minute experiment.
6. If something behaves unexpectedly, the program should terminate rather than continue.

Modifications to the code should ensure that these requirements are upheld.

In addition to these requirements, there were some general design philosophy considerations:

1. The particulars of controlling the amplifier should be defined separately from the user interface of sending control commands.
2. There should be tests to confirm that expected behavior is adhered to for things which do not require the amplifier.
3. The errors should be defined as their own classes, rather than Runtime or Value errors (built-ins for Python), to enable someone devising a high-performance experiment who knows what they're doing to appropriately filter out errors that may be tolerable in their unique circumstance if they want to use the ECI module only.

## 2 Module-Specific Design and Considerations

I endeavored to keep the docstrings informative enough, so this document focuses on how to think about the functinos written and modify them if needed, rather than the mechanics of how they work.

### 2.1 The NetStation Class

#### 2.1.1 Constructor

An IPv4 address and port number are required because that way the socket connection can be validated right away at construction, rather than potentially get deeper into experiment setup and then fail. We have to set the private connected attribute to `False` because we do not initially connect, and want to mark that. Validation of possible endian-ness values is also required so that we can fail early if an illegal one has been requested. By default, `'NTEL'` is used because almost all modern machines use little-endian byte ordering. We track the endianness with this abstraction because the ECI module does not track this for us (though it also assumes little-endian by default). Because no synchronization has happened and no recording has happened, we set the relevant attributes to `None` in order to make it clear that we haven't done anything yet.

#### 2.1.2 check_connected

This is a decorator to clarify which functions *require* that the amplifier is already connected. I copied this from a decorator tutorial. Note that it calls `args[0]` because in instance methods, `self` is always the first (and therefore 0th in index) element. Therefore, `args[0]._connected` means "for the decorated instance method, find this instance's connected private variable and check to see if it's `True`." We throw an error if not, because any method decorated this way requires that we be connected.

#### 2.1.3 connect

This method connects to a clock server. First, we enforce the constraint that the clock must be `'ntp'` or `'simple'`, which are the only clock sync methods that EGI provides. I had high and mighty dreams of adding in the simple clock, but found that the SDK documentation was incorrect on how to do this properly. As a consequence I never got around to implementing it because NTP works so well. If you wanted to implement a simple clock, you would need to keep a record of the last time you queried the clock within the instance, and then create a model of time (though that model may be that the clocks don't drift), and periodically update it with clock syncs using a new instance method. For now, the method connects to the amplifier and then sets the connected variable to `True`, and stores the NTP IP. Finally, it tells the amplifier what endianness to receive data in and instructs it not to do anything to mess with the clock, in anticipation of an imminent clock sync. It would be reasonable to go ahead and perform an NTPSync here, I truthfully don't remember why I chose not to do so.

#### 2.1.4 ntpsync

This method uses the NTPSync package to perform the NTP synchronization. It's important to make sure you send "Attention" to the amplifier before the sync. I never made sense of the server response, but if it responds in error it will crash. The system sends the amplifier a time, but it has to run through the `system_to_ntp_time` utility because the server *purports* takes NTP bytes, not a number or formatted datetime. At the end it creates a two new private instance variables, offset and syncepoch, to track the divergences of the clock.

#### 2.1.5 resync

This method just redirects to ntpsync because I could not figure out how the "resync" method was supposed to work, but we always got more inconsistent latencies than simply performing another NTPSync.

#### 2.1.6 resync_do_not_use...

I included this for completeness and it mirrors how `ntpsync` works, but it always performed worse. I included a "RESY" event send because I wanted to mark when it was attempted but that's not really required. I named it what I did because I wanted to be very unambiguous about using this method.

#### 2.1.7 disconnect

This sends the disconnect command to the amplifier, disconnects the socket, and marks the instance as not being connected. I would not recommend tinkering with this becuase it's simple and works.

#### 2.1.8 begin_rec

When you begin recording, this ensures an up to date sync. The simple clock branch is dead code because you can't do a simple clock sync in earlier-used methods. We mark the start of the recording time because NetStation actually uses this time to place events, rather than the supplied clock time, in all cases. Finally, we issue the command to the amplifier to begin the recording.

#### 2.1.9 end_rec

Sends the end recording command and unsets the recording epoch.

#### 2.1.10 send_event

This API allows the user to send events that were in the past, essentially. By default, it just records the current system clock time and uses that to set the time of the event. Because events have a large number of possible parameters, the method accepts many but has defaults for them intended to represent an event with the following attributes:

1. Start time of right now.
2. Duration of 1 millisecond.
3. A type of all spaces.
4. A label of all spaces.
5. A description of all spaces.
6. No associated data.

Note that you can send a relatively complex dictionary of data, but there are restrictions in the notes. This is ultimately sent to the ECI function `_package_event`, which converts the data into an appropriate byte string to accompany the command. The command is sent, and no instance variables are marked because events don't change any internals. NOTE: sending events too quickly in succession will cause a connection reset. Some researchers do not consider this and may think the program is buggy. It may be worth your time to create something idiot-proof, like setting up a try-except around the command sent to catch any connection resets and report that the user is being not-too-smart, but in a tactful way.

#### 2.1.11 rec_start

Just returns the start of the recording. Consider converting to a property and adding some behavior for what happens if the method is used outside of a window that is being recorded. This was implemented mostly for debugging and is potentially dead code.

#### 2.1.12 since_start

I can't remember why this should not be used. It should probably be considered dead code and removed.

#### 2.1.13 _command

Private method to send commands to the amplifier. The eci module's `build_command` function will convert it all into bytes. Pay careful attention to make sure that any changes in this function mirror changes in `build_command`, and vice versa. With the command built, it sends the message over the socket. The socket response is parsed, and will trigger an error if the amplifier responds in an unexpected way (as handled by eci module's `parse_response`).

#### 2.1.14 TODO notes

There are a bunch of pieces of code commented `TODO: turn into a debug option`. These are pieces of code I was using to debug what I was sending to the amplifier and how it was replying. You might consider creating some sort of debug mode to use NetStation in, and re-enabling these print statements. It will be especially useful if you need to add a new feature or diagnose an unexpected behavior.

## 2.2 The eci module

### 2.2.1 preamble

There are a few variables declared at the top to apply to the whole module. `blue` and `reset` are definitions for printing colored ASCII text. Currently this is dead code, but if you implemented a debug mode they could be useful. The variable `byte_table` maps each command to the appropriate command byte to send. The variable `requires_data` indicates which of the commands (each of which is also in `byte_table`) require data. NOTE: `NTPReturnClock` command requires data, though the SDK documentation indicates this is not the case. The variable `allowed_endians` indicates which endian types are allowed. NOTE: if you ever design something requiring endians, please use "big" and "little" instead of "every platform is special but actually maps to just two behaviors." This is the sort of behavior most rational developers would expect. The variables `INT_VAL_*` are there because of a weird Python quirk. If you index into a byte string, an `int` is returned. As a consequence, I made these hardcoded values to easily map any integer value returned to a byte. The variable `MPS` is just to avoid magic numbers; it's the number of milliseconds in a second.

### 2.2.2 build_command

This function essentially converts a command specification into the bytes to be sent to the amplifier. First, we check to make sure that the command string is valid. The variable `tx` tracks the current byte string, and is what ultimately gets transmitted to the amplifier. Then we perform checks to make sure that the command is matched to whether data is expected. Then, if data is required, we validate it and convert it to bytes as needed. In the case of `EventData`, no validation is performed. Use the `package_event` (Section 2.2.4) function to package data before sending it to this function. A TODO notes that `package_event` could be added to the command builder in order to guarantee this two-step process, but it would complicate the API slightly and I never got around to it. It may be worth doing if you'll use the ECI functions by themselves, but the NetStation object encapsulation handles this for you pretty well. Finally, we return the data to be transmitted as a byte array. Error granularity is relatively fine, allowing you to easily check what a user who isn't paying attention did to trigger an error. Use Table 1 to guide sending data with commands. One note is that this was done as a first pass. You could refactor this to have each command be its own function, which would simplify the logic. You could also keep this function to leave the NetStation object with the same calls, but then use a dictionary mapping each commmand to function pointers, to simplify the logic.

| Command | Data Required |
|---|---|
| `Query` | `endian in ("NTEL", "MAC-", "UNIX")` |
| `NewQuery` | None |
| `Exit` | None |
| `BeginRecording` | None |
| `EndRecording` | None |
| `Attention` | None |
| `ClockSync` | `int` number of milliseconds (unsure) |
| `NTPClockSync` | `float` seconds from `time.time` |
| `NTPReturnClock` | `float` seconds from `time.time` |
| `EventData` | `bytes` representation of data |

Table 1: A tabel of commands and the data they require.

### 2.2.3 parse_response

This function takes a response byte array and converts it to a Python object. This is best expressed in table form. This was set up mostly to trigger errors from a consistent function. There may be a better way of organizing this. Some server responses are totally undocumented. For example, replying "1" is not mentioned anywhere but happens sometimes. Al-

| Response | Meaning |
|---|---|
| `Z` | Fine, have a timestamp |
| `I` | Fine |
| `I` | Fine, here's my version number. |
| `F` | NOT Fine |
| `R` | NOT Fine and it's related to recording |
| `1` | Fine |
| `S` | Fine, have a timestamp |

Table 2: Various server response starts. Note that "I" occurs twice. This is because sometimes it comes by itself, and other times it comes with a version number, too.

most as infuriating, the SDK server simulator replies with an 8-byte NTP timestamp. However, the real amplifier responds with either of `Z` or `S`, followed by the NTP timestamp. This function looks confusing because I was confused while writing it. Essentially, it returns `True` if everything is fine and the response contains no data. If the response contains data, it seems to only ever be a timestamp, which is the time in the NTP epoch according to the amplifier!.. WHICH IS WRONG. The *actual* NTP epoch starts in 1900, but the amplifier starts at the *UNIX* time epoch of 1970. At least, I think. That made the timestamps make much more sense. Good luck if you have to tamper with this function. Table 2 indicates my best guess as to possible responses and what they mean.

#### 2.2.4 package_event

This function is relatively well-documented in the docstring and I feel it works well. The main thing to note is that when data is added to the bytestring, it tracks the length of the total byte string because the total number of bytes needs to be sent as part of the datagram. It causes bizarre behavior if this value is wrong, or just triggers a server failure.

### 2.3 The util module

I feel that this module is actually pretty well-documented. The big thing to note is that NTP timestamps are 4 bytes representing the integer number of seconds since the epoch start, and 4 byte representing the number of increments of $2^{-32}$ seconds since the last second. `sys` implies system time, usually from `time.time`, and `ntp` implies NTP time.

#### 2.3.1 socket_wrapper

This defines the `Socket` class, which is a simple wrapper for the Python socket. I believe Pete started this but I can't remember anymore. The key things to note are that you can modify the buffer size for reading and writing. You can also modify the timeout, which is in seconds. I just set it to 1 second but maybe there's a better approach to picking a good timeout.

## 3 Testing

### 3.1 pytest tests

Run the test suite with

```
poetry run pytest
```

if you have poetry installed. Alternatively, if you don't, enter `egi_pynetstation/` and run

```
pytest tests
```

to run the tests. I don't recall much of the test writing, but they are mostly designed to ensure that the tests replicate the expected API behavior. They do NOT check to make sure that the behavior at the amplifier is correct!

### 3.2 amplifier tests

Amplifier tests should be done with Pete. In general, I have uncommented parts with `TODO: make debug option` while running amplifier test, but you could presumably create a debug mode that would print this stuff. A simple test where you tell PsychoPy to change the color and send an event, and then plug a photodiode into the server computer, should be adequate to do basic timing.