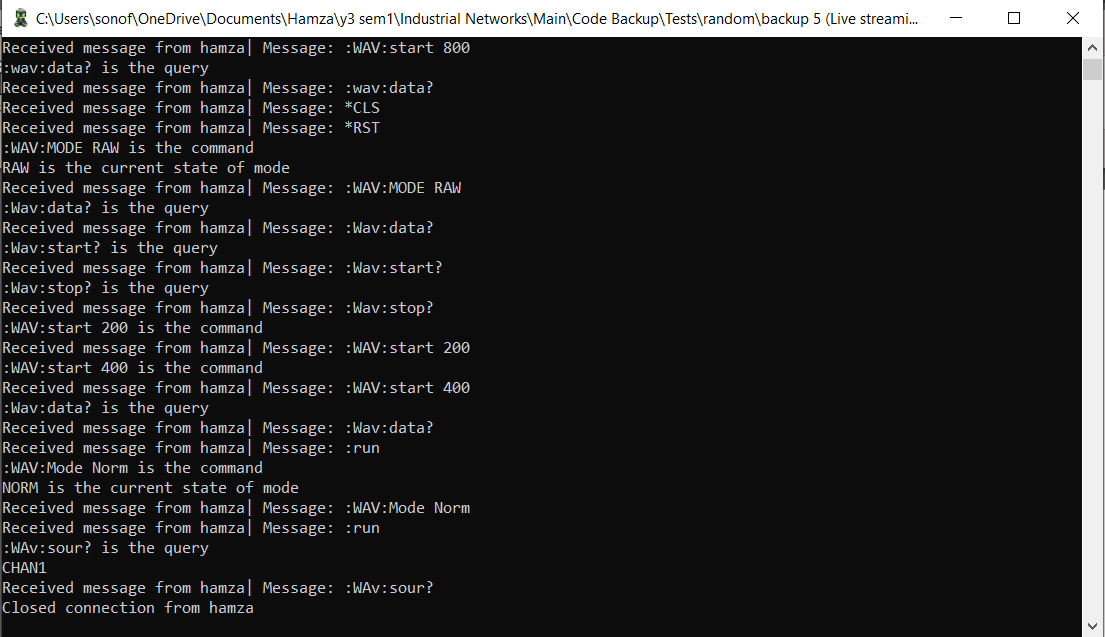
Overview

(Note: The test cases are defined after the Overview section, which can be skipped)

The **server** is a console-based application:



It is a multithreaded application with three threads running:

* **Wave-Generation Thread**: Generates the (random) square wave for channel one and provides end-points to access different parts of the wave (e.g. start, stop points, xref etc.). It also creates an (simulated) **internal memory** represented by **json files** (storing 1320 points), storing wave information beyond the range of the array storing 120 points.
* **Data Streaming Thread**:- Streams the square wave voltage data via a **UDP socket** to the client.
* **Main Thread**: The main server logic. Accepts commands from the client via a **TCP socket** and uses **regex** to recognize command pattern to separate the types of commands and then detect individual commands via conditional statements. Uses **state machines** to manage states of different properties (e.g. Channel, Mode etc.) and handles any condition of mutually exclusive states.

The **Client** is a GUI-based, multithreaded application (though the main thread does not have a GUI).

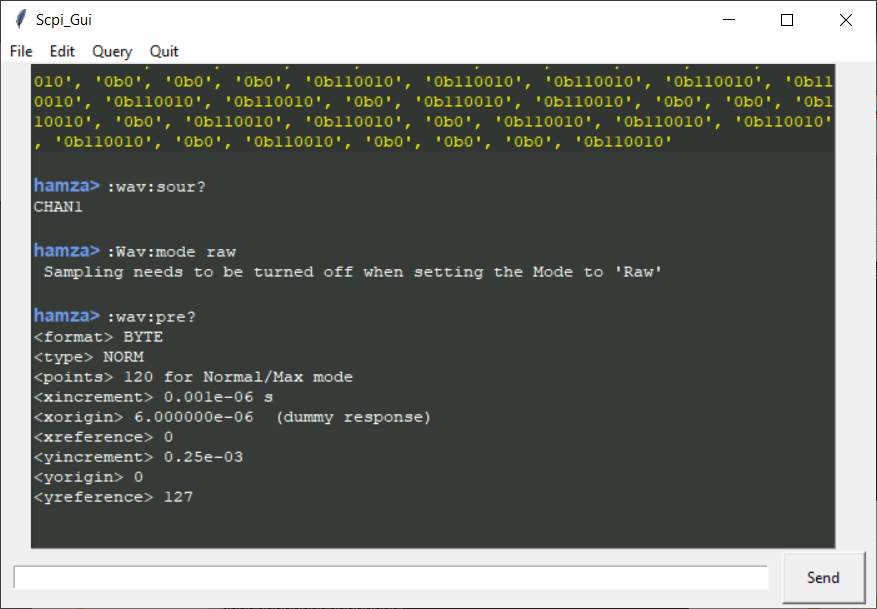
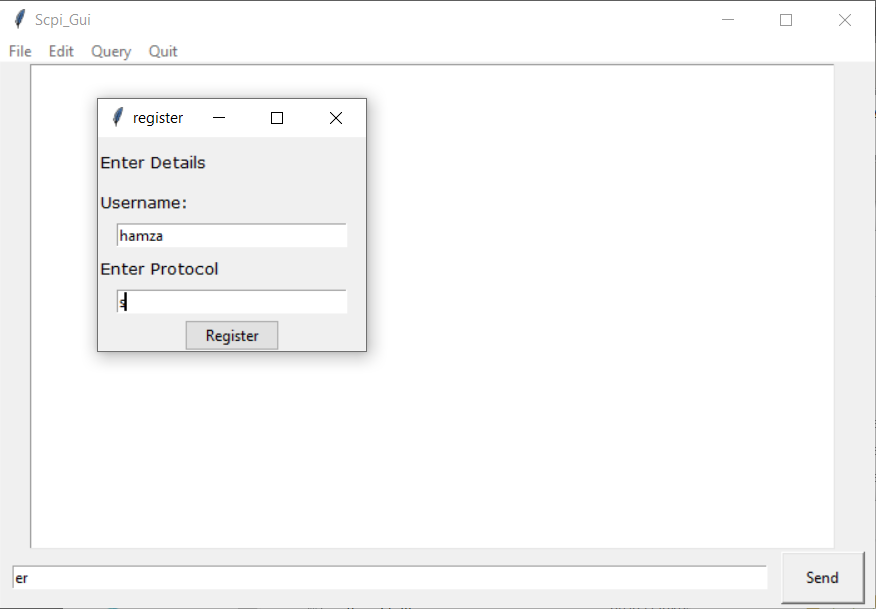


Figure 1 (Client-Window-1)

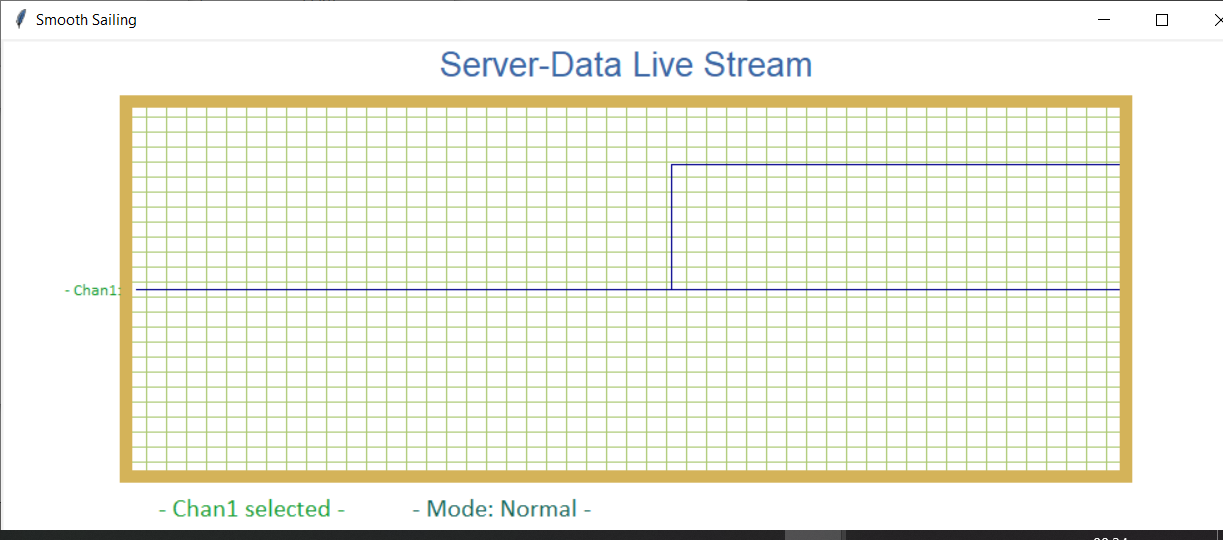
The client has two GUI windows, the first window shown above is where the commands are entered, to interact with the SCPI server.

The first time the application is opened, and the user enters a command, the user is prompted with a popup window requesting for a username and the protocol.



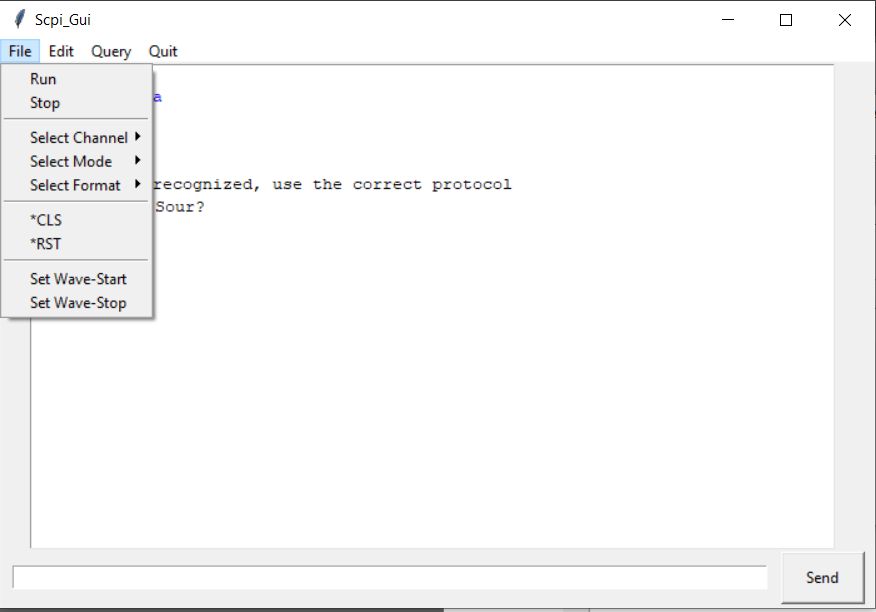
Based on the chosen protocol (S (short) or L (long)) the user then is supposed to stick to the chosen protocol when entering the commands.

The second Window is the live-data-streaming window which shows the wave-generated via the Wave-Generation thread on the server side, in real time, transmitted via udp-socket, broadcasting the voltage data of the square-wave.



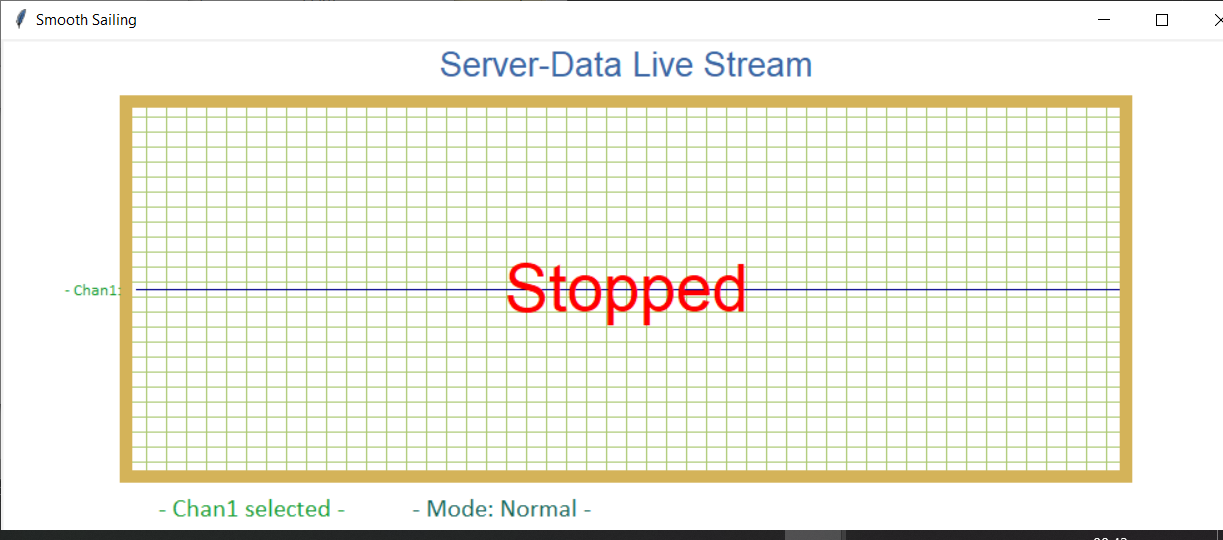
The window also shows the selected channel and the Mode, with the wave-data always saying chan1, because other channels aren’t connected (technically not been coded to generate wave). The channel and Mode info might not be accurate if the client was closed while the server was running and the server is not in the default state when the client is re-opened (however, as soon as the chan or mode is changed again, it will update on the second window).

The user is also provided with a menu system which lets the user do almost everything that he/she could do with the command line.

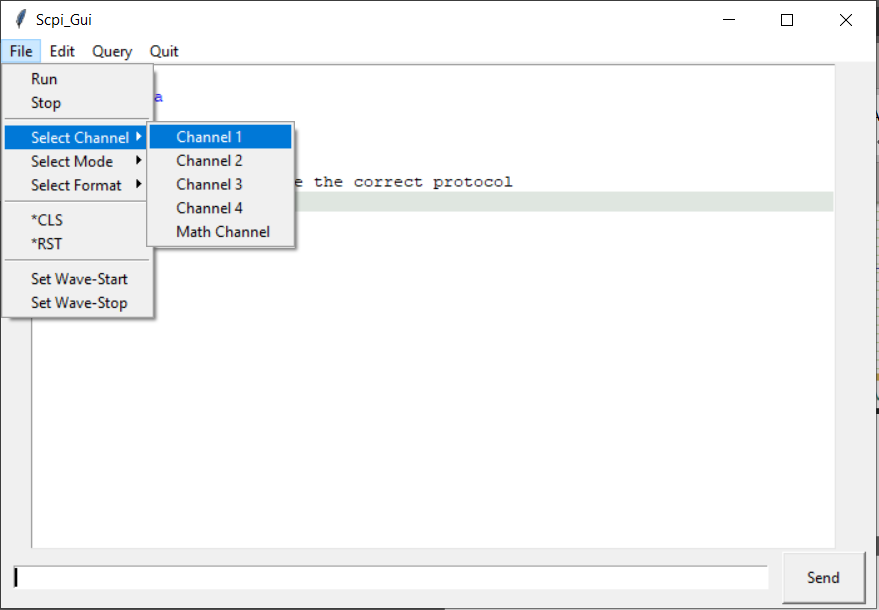


The **File menu** provides the following options:

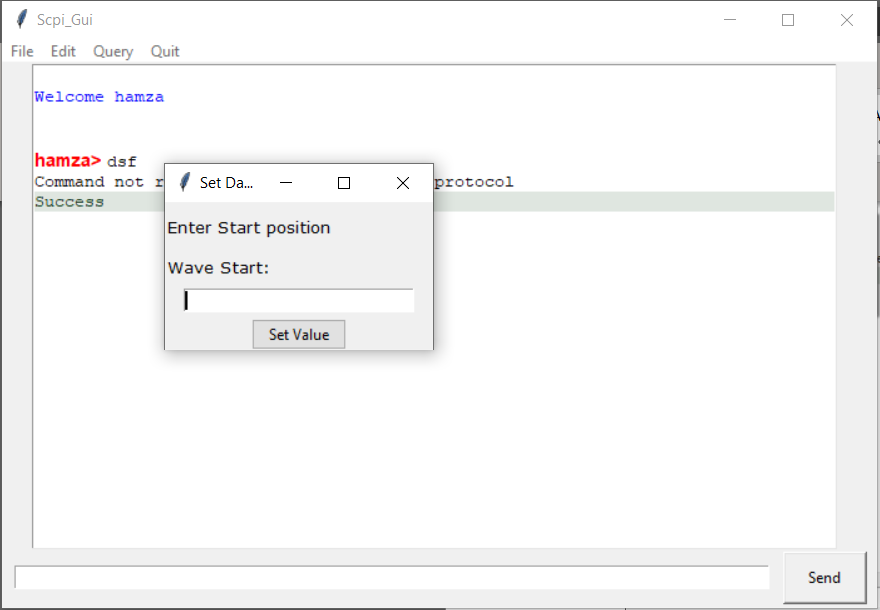
* **Run**: To have the oscilloscope in the running state, so essentially the wave is being generated (for just channel 1 in the current version) and the live streaming is happening.
* **Stop**: To stop the wave generation and the live streaming. When the this option is used, it will be evident on the live-streaming screen:



* **Select channel** will let the user select the channels, **select mode** will let the user select and same for **format**:



* **\*CLS** would clear the event registers (doesn’t really affects anything in this version but is needed to be used before data is retrieved for the first). **\*RST** sets channel, mode and format to the default state is needed to be used before data retrieval.
* Set **Wave-start/stop** would prompt the user to enter a number for wave-start and wave-stop:



The **Edit** allows the user to chose an appearance:

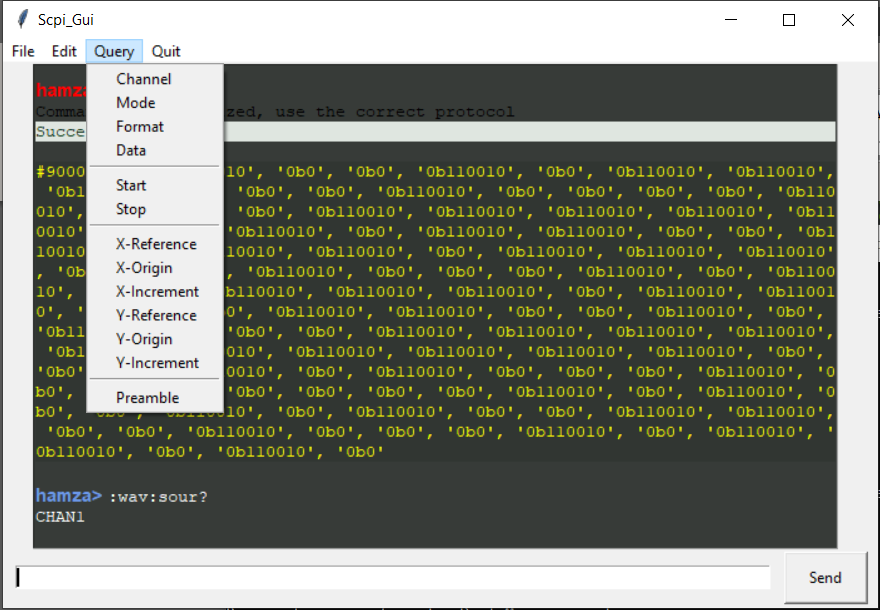
* Default or
* Dracula



Figure 2 (Dracula)

(Note: The previously entered text would not be re-rendered when changing appearance)

The **Query** menu would allow the user to query everything the user can query via the command line:



Finally, Quit allows the user to quit.

(Note:

*It is very important that Quit option is used to quit the client application because the main thread does not have a GUI so exiting child thread wouldn’t close the main thread and would need to be killed via Task\_manager before the client application can be opened again because of the same socket being used again.*

*This is an acknowledged design flaw.*

*The quit menu is connected to a sys.exit command in the main thread which quite all other threads as they are daemon threads*.)

The problem of real-time updating the screen with the reply from the server in the main client thread was solved by using non-blocking tcp socket. The socket is made blocking as soon as the command is sent but as soon as a reply is received, the socket is set back to non-blocking, this allows for asynchronous communication between client and the server but at the same time preventing any errors arising from logic-handling before a reply is received.

Test cases

File Structure:

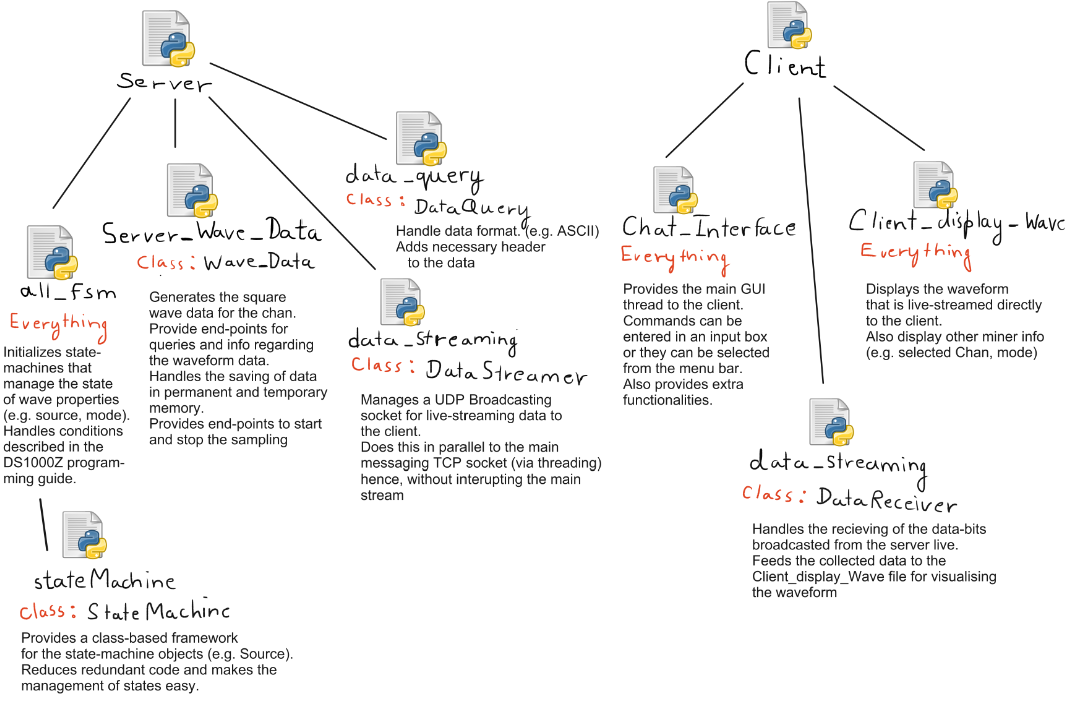


Figure 3 (File Structure)

* **Test Case 1:** Implement the following commands on the stub-server, with the server responding with a suitable dummy response

:WAVeform:SOURce

:WAVeform:MODE

:WAVeform:FORMat

:WAVeform:DATA?

:WAVeform:STARt

:WAVeform:STOP

Sampling, Source, Mode and Format have their own objects based on the stateMachine class:

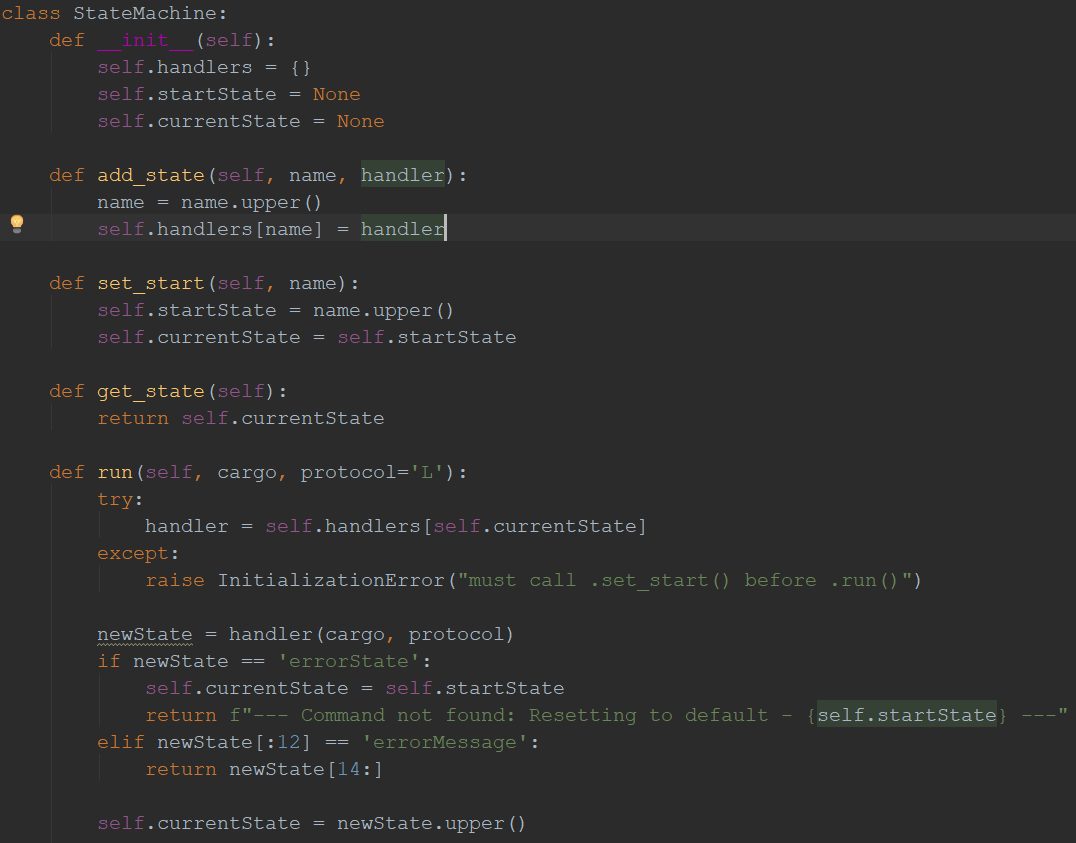
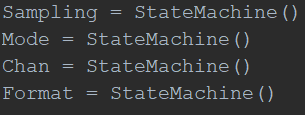


Figure 4 (State Machine Class)



* 1. **WAVeform:SOURce**

In the all\_fsm.py file, the channel\_transition() function statement handles the DS1000Z requirements for Source:

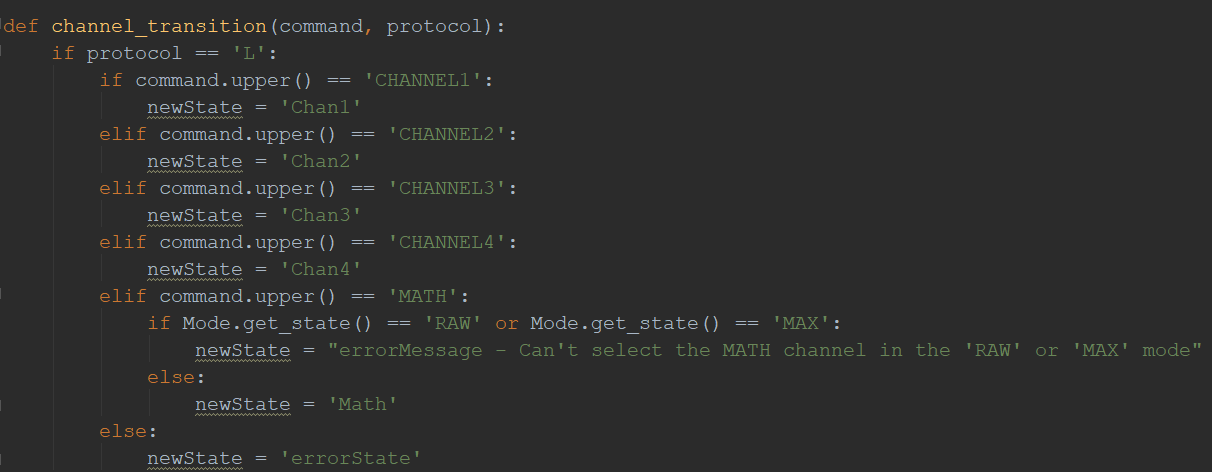
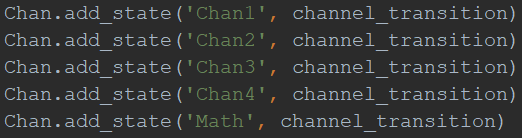


Figure 5 (Channel Transition Function (extends for the small protocol))

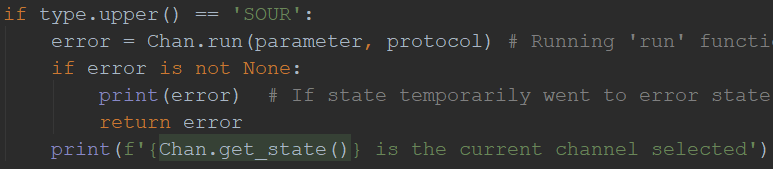
Initiated in the server.py file:



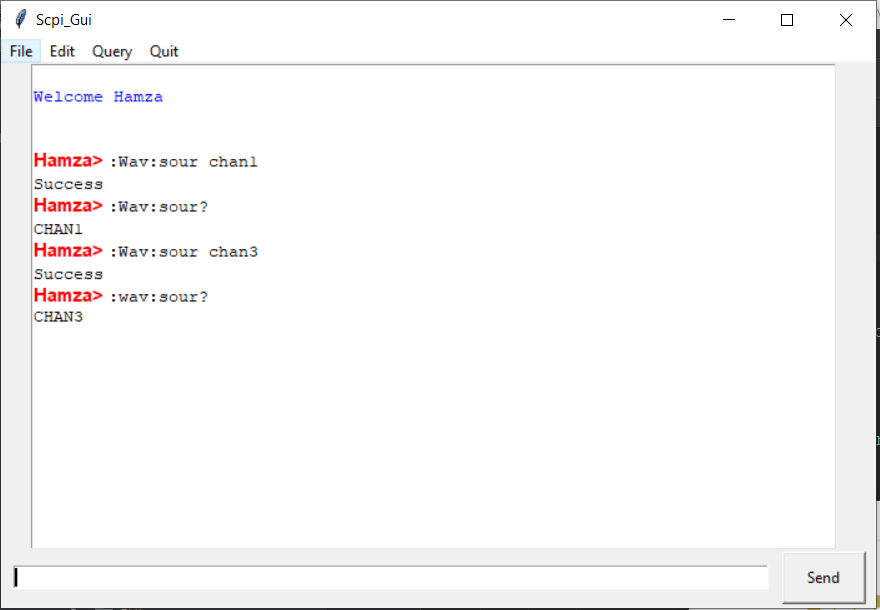


Entered command handling is done via regex to recognize the command pattern:  


The specific commands options are handled later:



Demonstration:



* 1. **WAVeform:MODE**

In the all\_fsm.py file, the mode\_transition() function statement handles the DS1000Z requirements for Mode:

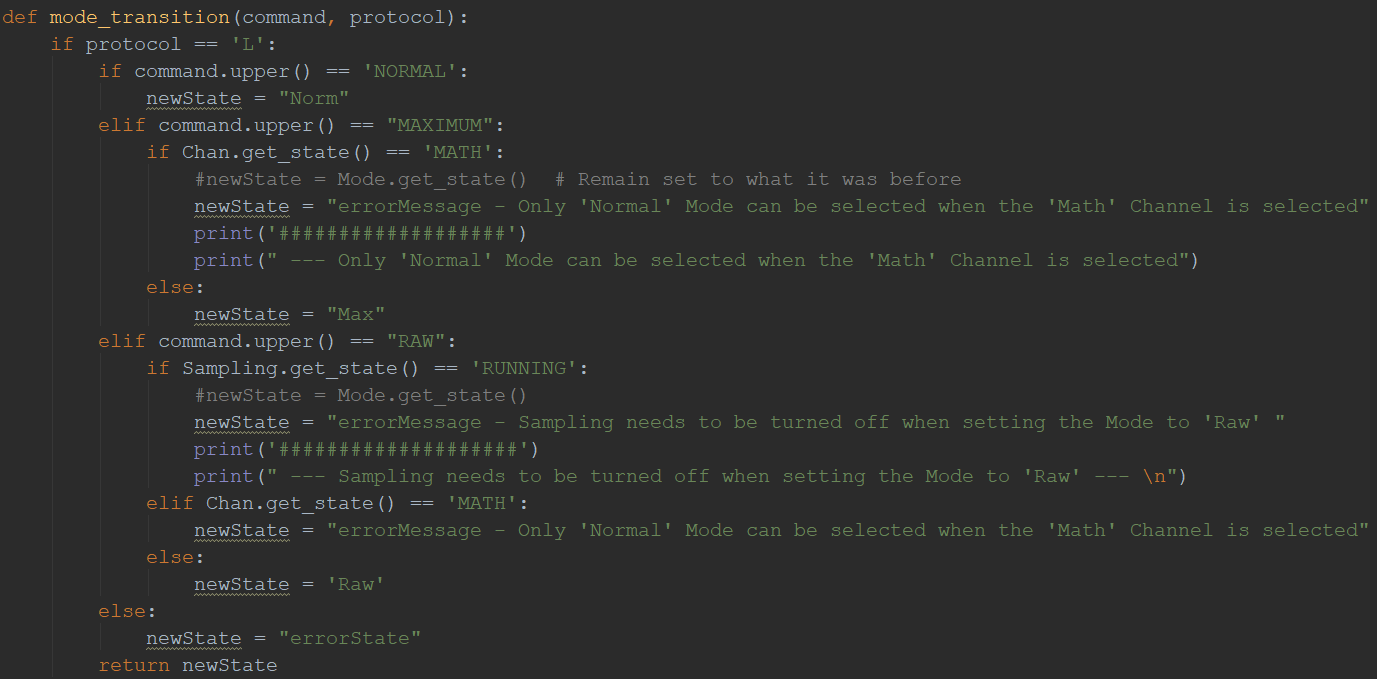
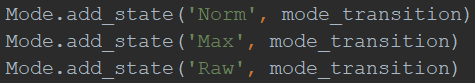


Figure 6 (Mode Transition (extends for the small protocol)))

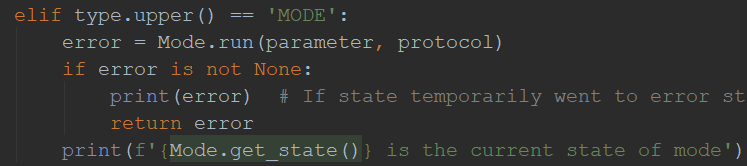
Initiated in the server.py file:



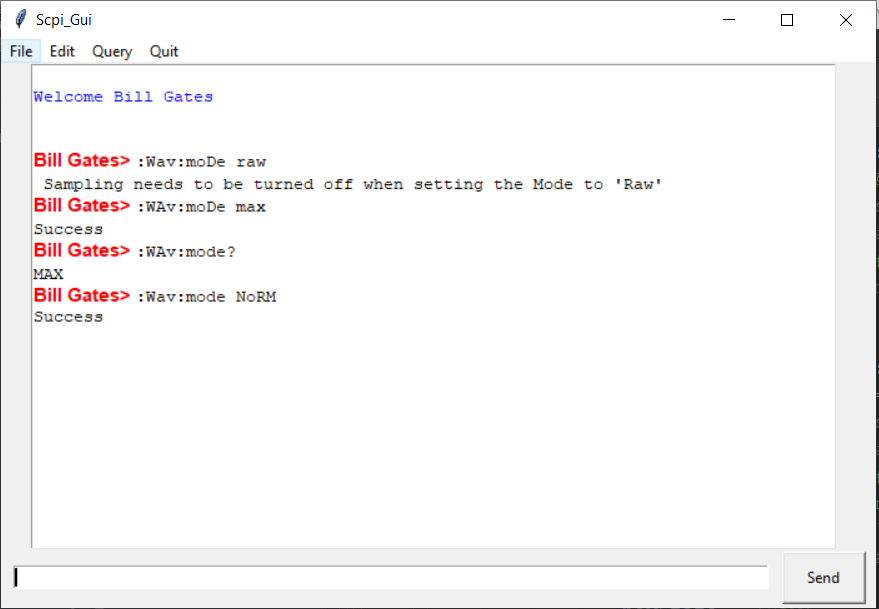


Entered command handling is done via regex to recognize the command pattern:  


The specific commands options are handled later:



**Demonstration:**



* 1. **WAVform:FORMat**

In the all\_fsm.py file, the format\_transition() function statement handles the DS1000Z requirements for Format:

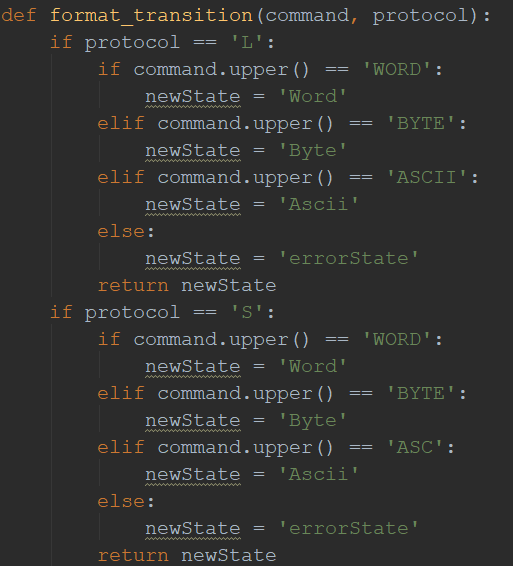
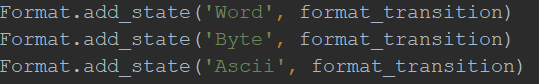


Figure 7 (Format Transition Function)

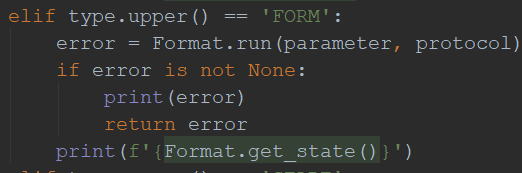
Initiated in the server.py file:



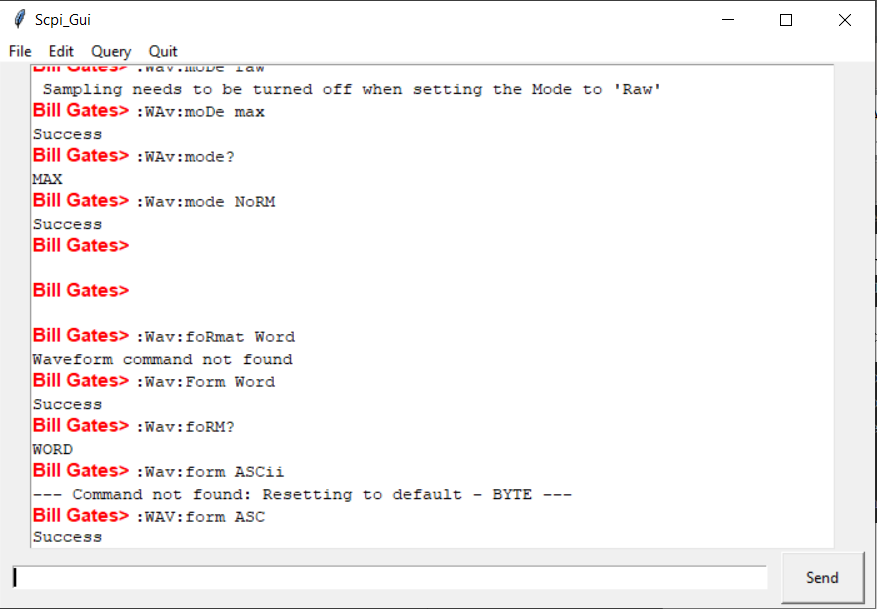


Entered command handling is done via regex to recognize the command pattern:  


The specific commands options are handled later:

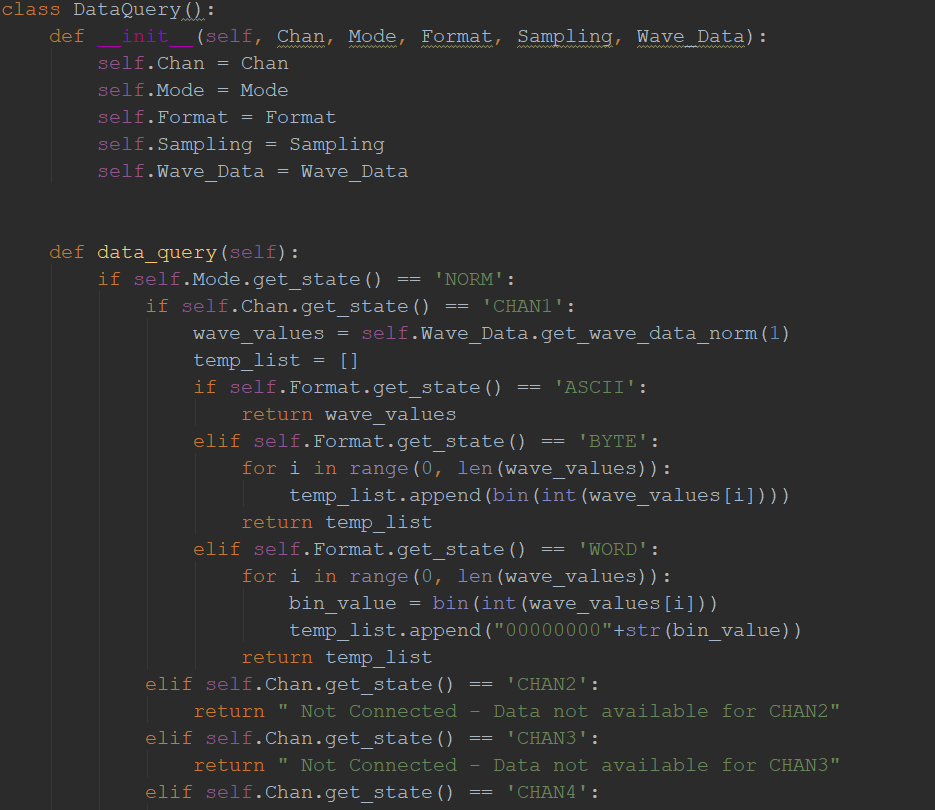


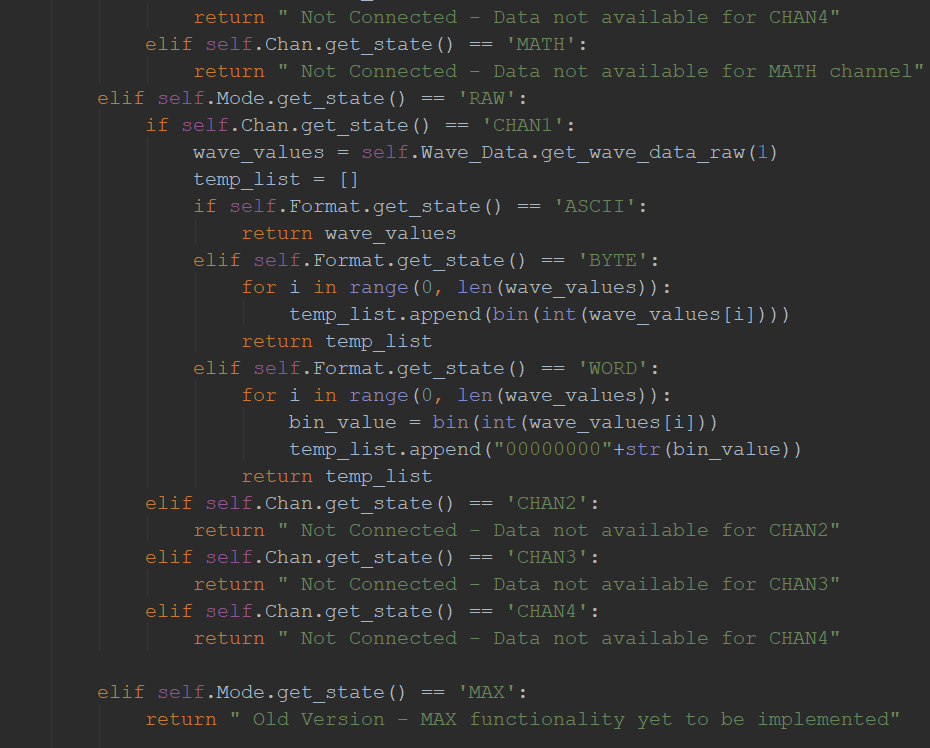
**Demonstration:**



* 1. **WAVeform:DATA?**

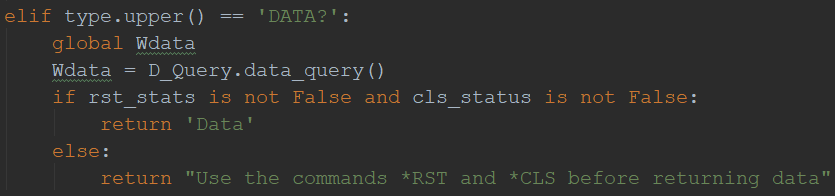
Data is first generated in the Server\_Wave\_Data.py file, the handling of the data is query is mainly done in the data\_query.py file:



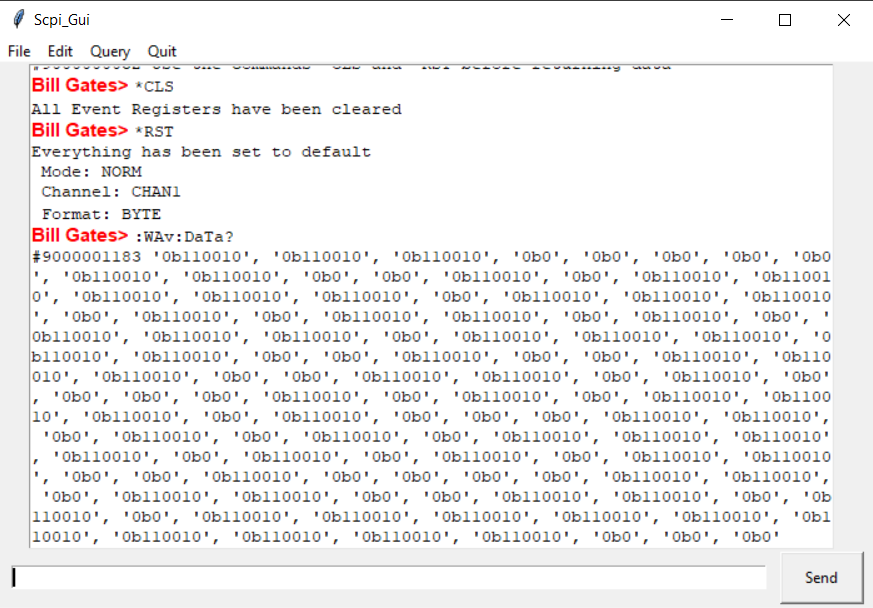


Entered query handling is done via regex to recognize the command pattern:  


The specific commands options are handled later:



**Demonstration:**

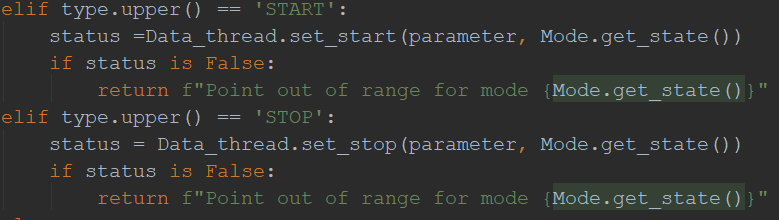


* 1. **WAVeform:START and WAVeform:STOP**

The entered command is handled in the server.py file

Entered command handling is done via regex to recognize the command pattern:  

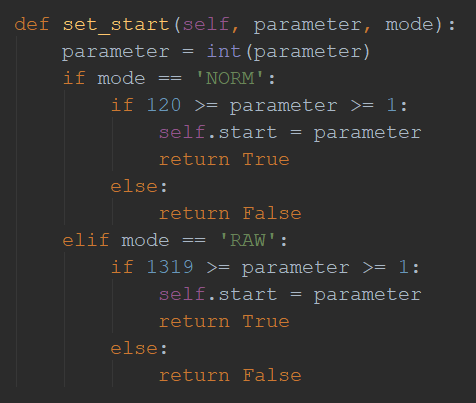

The specific commands options are handled later:

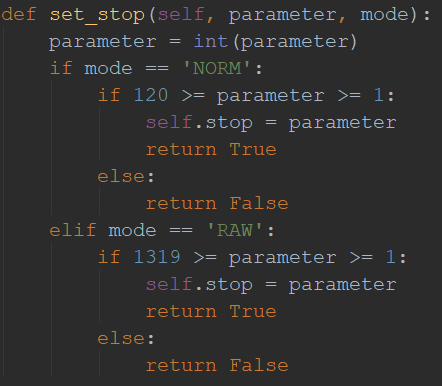


The ‘parameter’ passed into the *set\_stop* or *set\_start* functions is the value passed for the :WAV:START or WAV:STOP (e.g. :WAV:START *149*). The command is split based on the space between the value and the command.

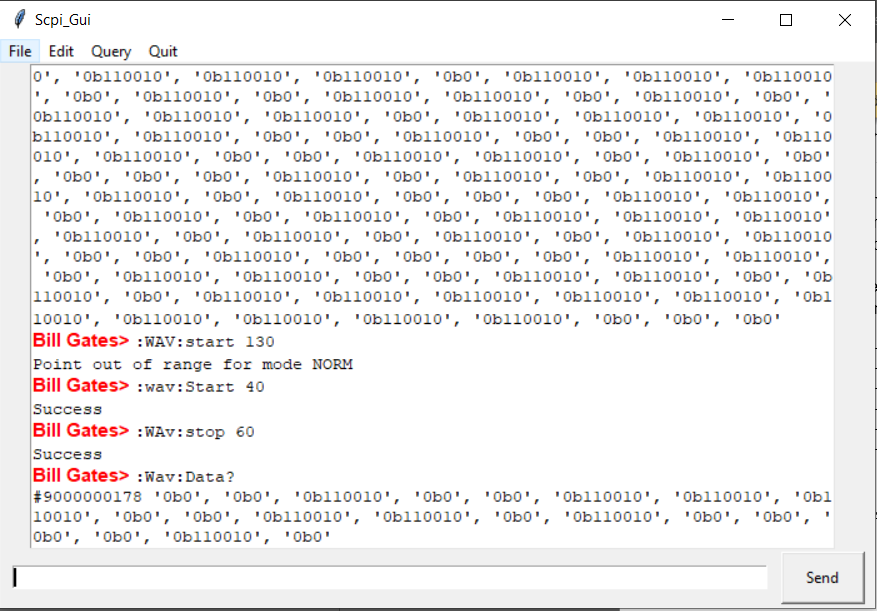


The ‘Data\_thread’ which is the Wave\_Data object (for generating and storing the wave data) has two class functions to set the start and stop points to when accessing a data segment:





**Demonstration:**

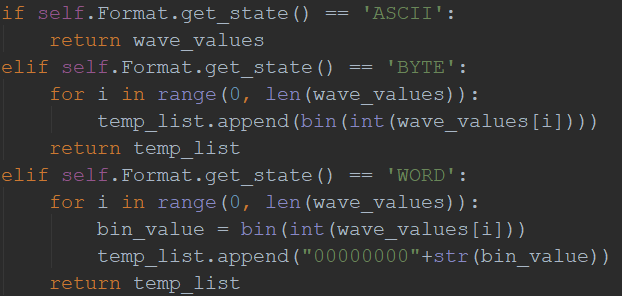


* **Test Case 2:** The ability for the client to use the “ASCII” form of the “:WAVeform:DATA?” command to retrieve data from the stub-server.

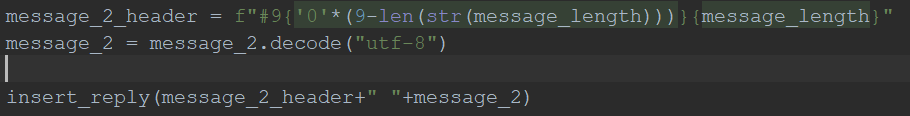
(Also shown above in the Data query section)

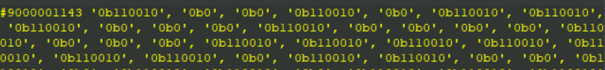
The data by default is ASCII, though as per the requirement BYTE is made default by using the bin() function on every single data point in the wave.

When the data is to be converted to ASCII, bin() function is *not* applied to the wave data points:

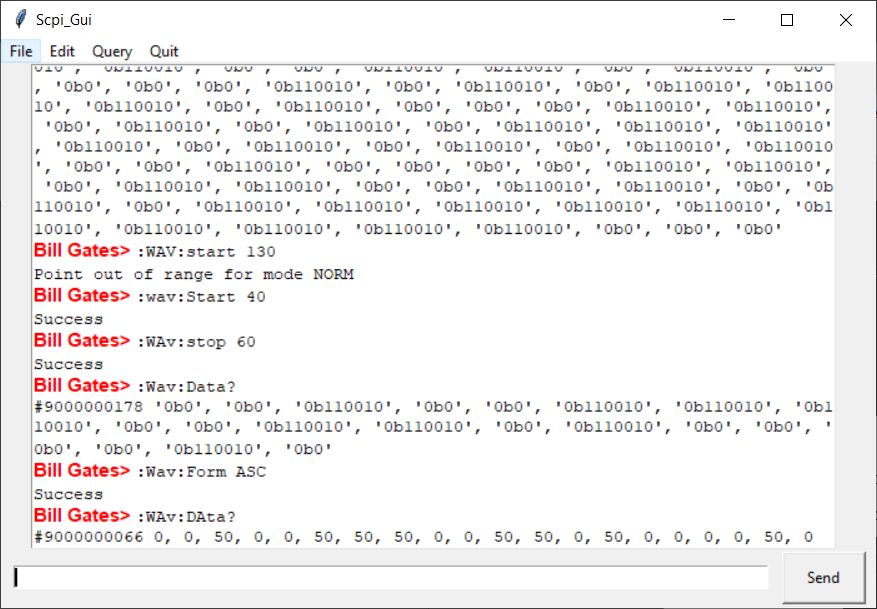


Before printing out the data to the client GUI, the length of the data is added as the header to data-points with signature pattern starting with ‘9’.





**Demonstration:**

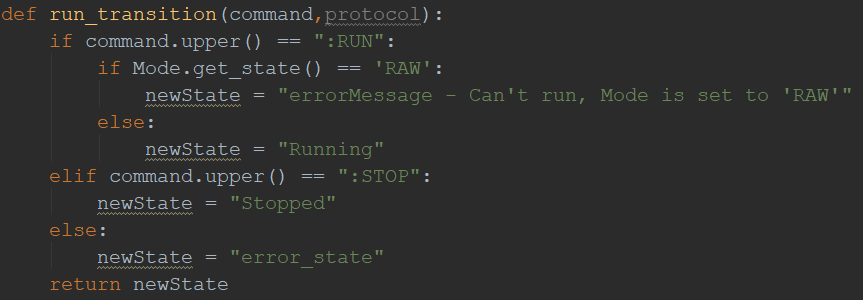


* **Test case 3 -** The ability for the client to issue the following command sequence to the stub-server, and for the stub-server to respond appropriately

|  |  |
| --- | --- |
| Command | Expected Action |
| :STOP | Set the instrument to the STOP state |
| :WAVeform:SOURce CHAN1 | Set the channel source to CH1 |
| :WAVeform:MODE RAW | Set the waveform reading mode to RAW |
| :WAVeform:FORMat ASCII | Set the return format of the waveform data to ASCII |
| :WAVeform:STARt 1 | Set the start point of waveform data reading to the first waveform point |
| :WAVeform:STOP 120 | Set the stop point of waveform data reading to the 120th waveform point (the last point) |
| :WAVeform:DATA? | Read the waveform data in the internal memory (all the points) |

**3.1 :STOP**

The state of sampling is handled by the ‘Sampling’ state machine with the following function handler:

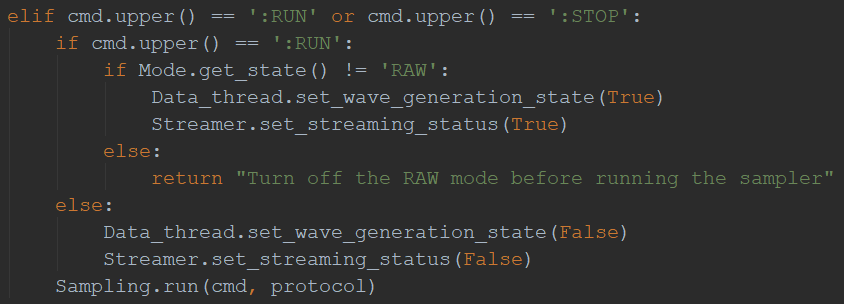


Initialized in the server.py file

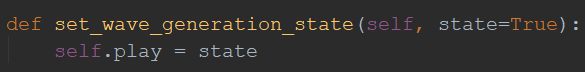




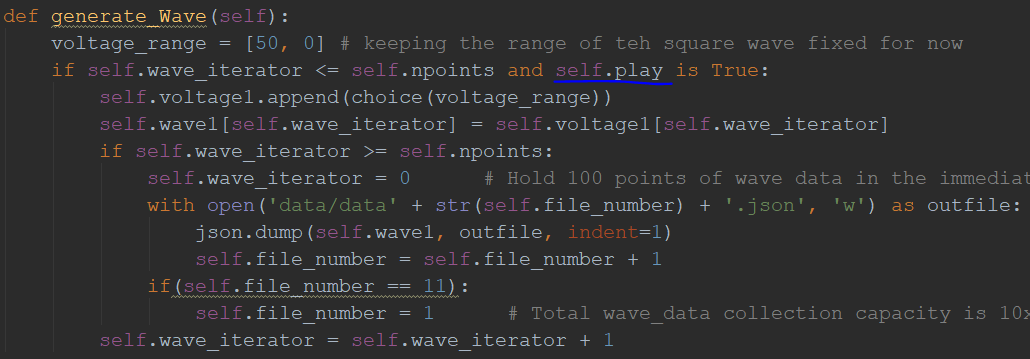
The command handling is also done in server.py file:



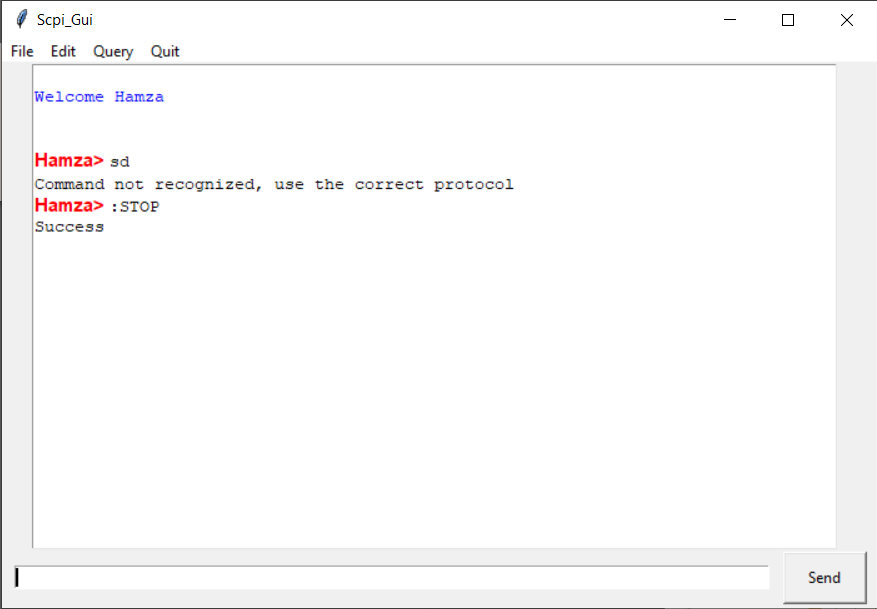
The ‘:RUN’ command passes ‘True’ as the parameter to ‘set\_wave\_generation\_state()’ function which is a class function of the Data\_thread object (from Wave\_Data class for generating and storing wave data points).

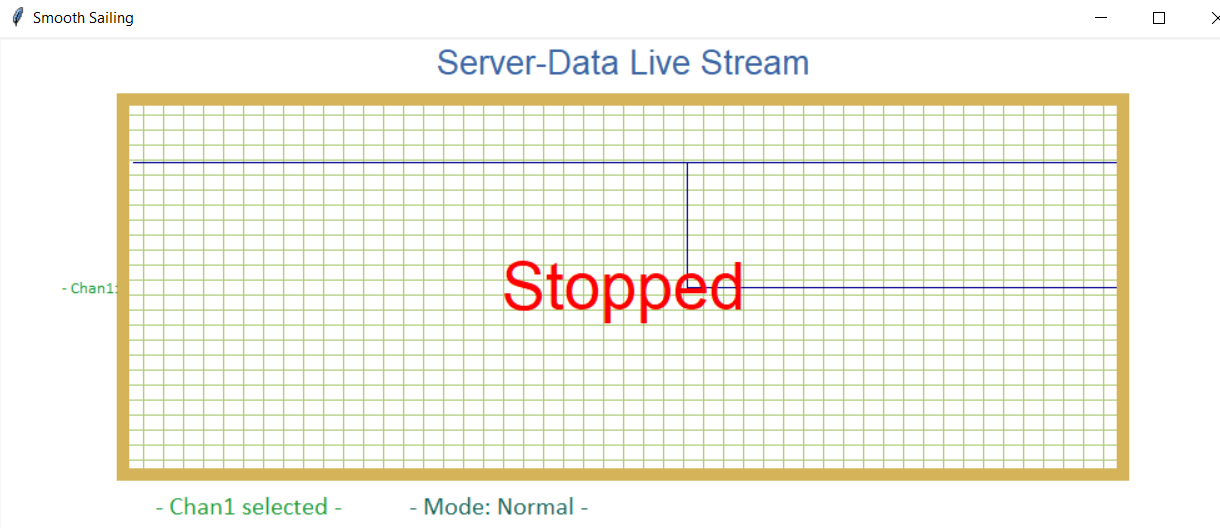


‘self.play’ then determines whether the Wave for the selected channel should be generated (Sampled) or not:



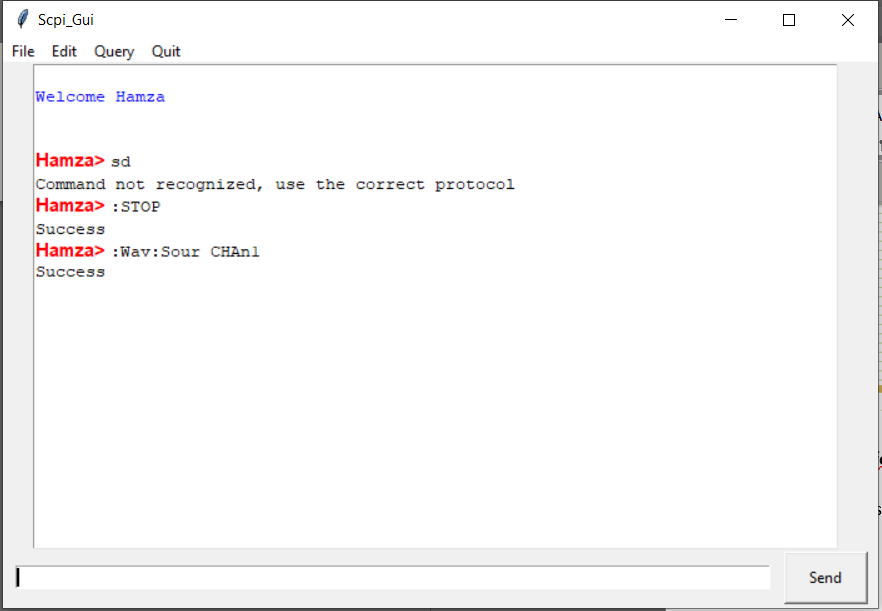
**Demonstration:**





**3.2 :WAVeform:SOURce CHAN1**

Demonstrated in detail in section 1.1

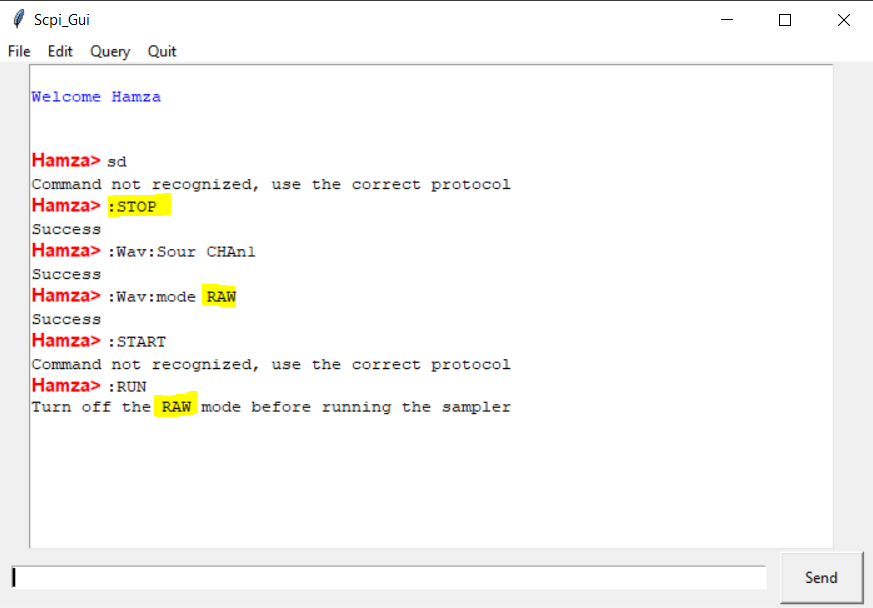


**3.3 :WAVeform:MODE RAW**

Demonstrated in detail in section 1.2

Since the sampling was turned off, raw mode could be selected.

To turn ON the sampling, raw mode would need to be turned off

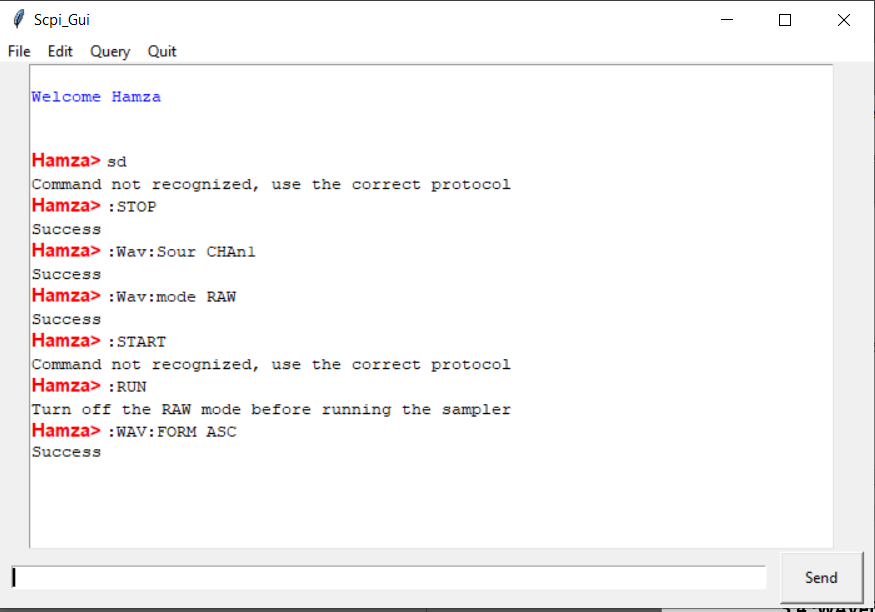


Server side:

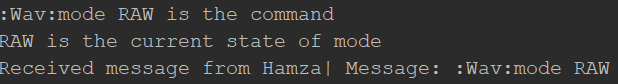


**3.4 :WAVeform:FORMat ASCII**

Demonstrated in detail in section 3.4

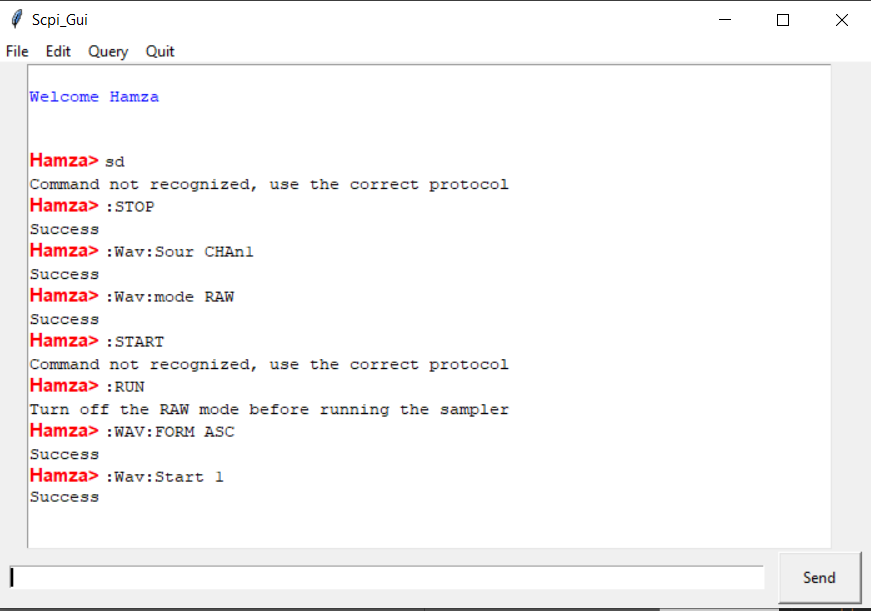


Server side:



**3.4 :WAVeform:START 1**

Demonstrated in detail in section 1.5

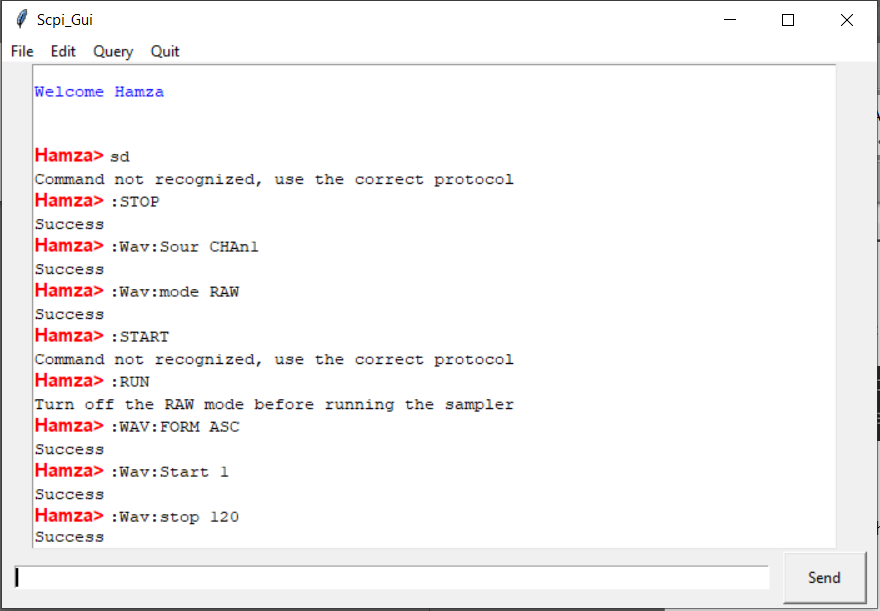


Server side:



Start point has been set to 1 on the server side.

**3.5 :WAVeform:STOP 120**

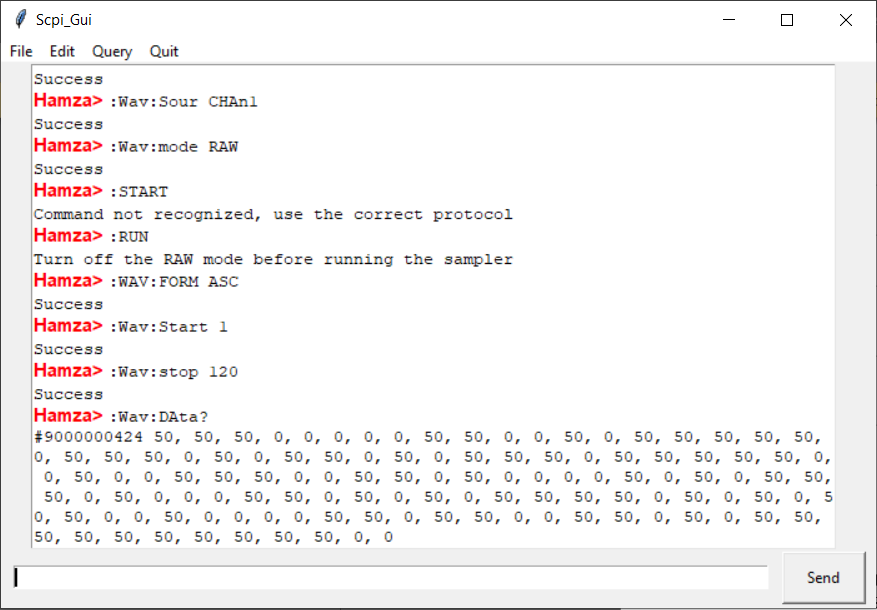




The Wave stop point has been set to 120.

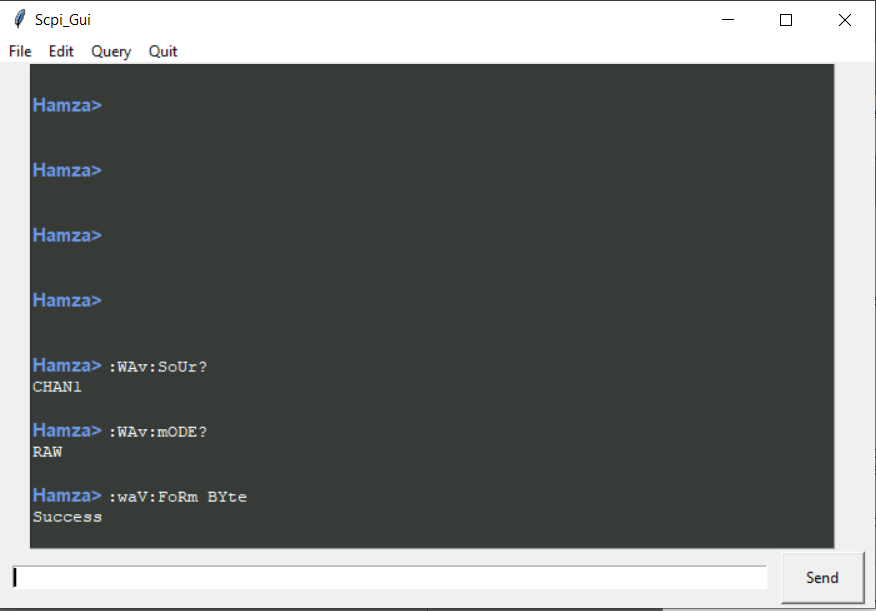
**3.6 WAVeform:DATA?**

All wave-points recorded in the range 1-120 before the sampling was turned off, are printed.

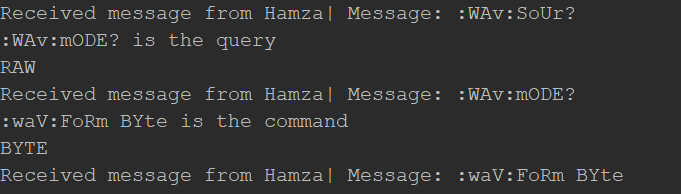


**4 Case Insensitive**

The application is case insensitive



Server side:

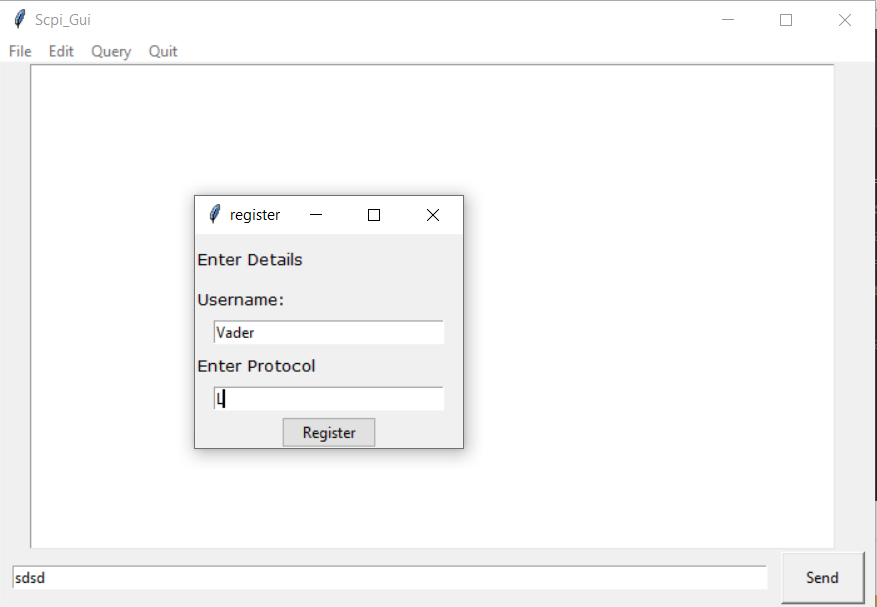


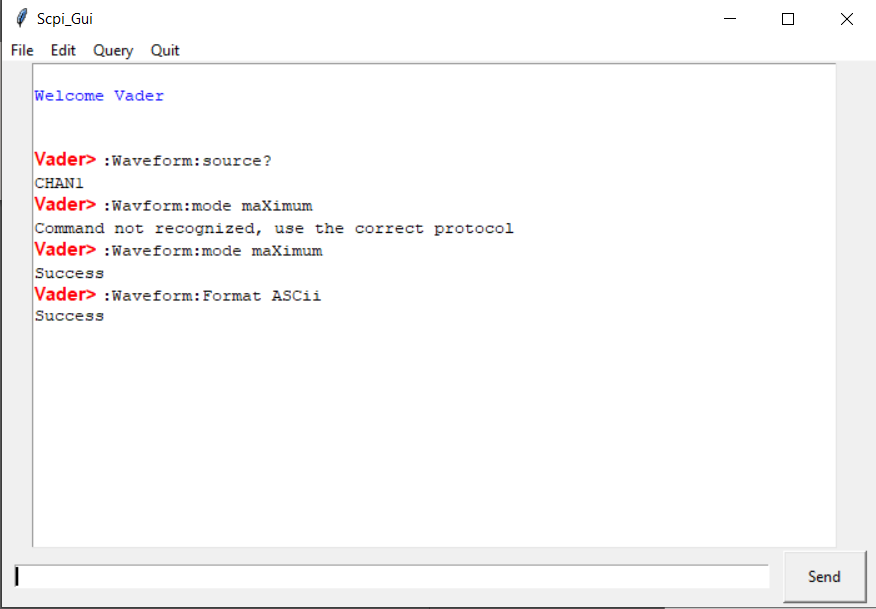
**5**

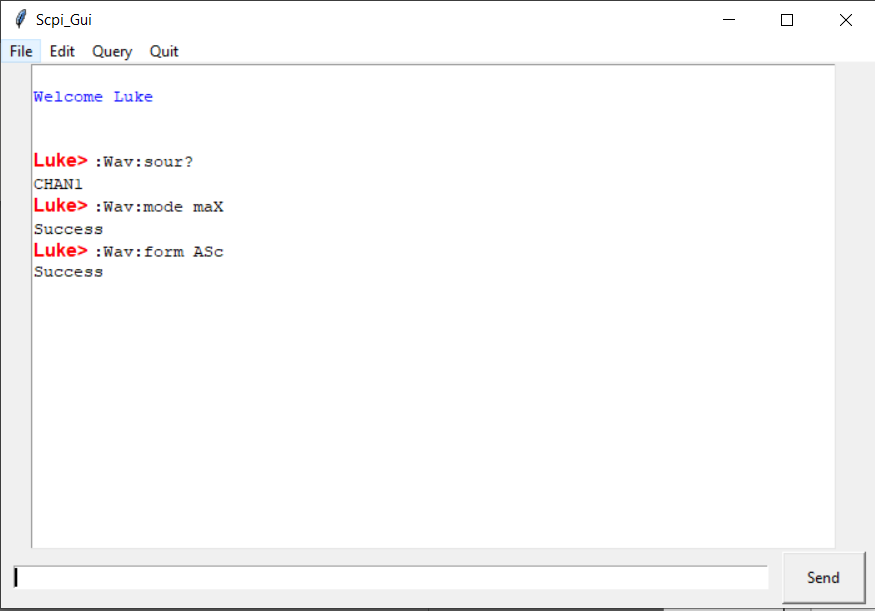
**The SCPI client can choose to use either the long-form or the short-form of the protocol command, e.g. “:WAVeform:FORMat?” and “:WAV:FORM?” are identical.**

When the program is first started, the user would be given the option to chose ‘L’ long protocol or ‘S’ short protocol.

Based on this, both type of commands can be used:



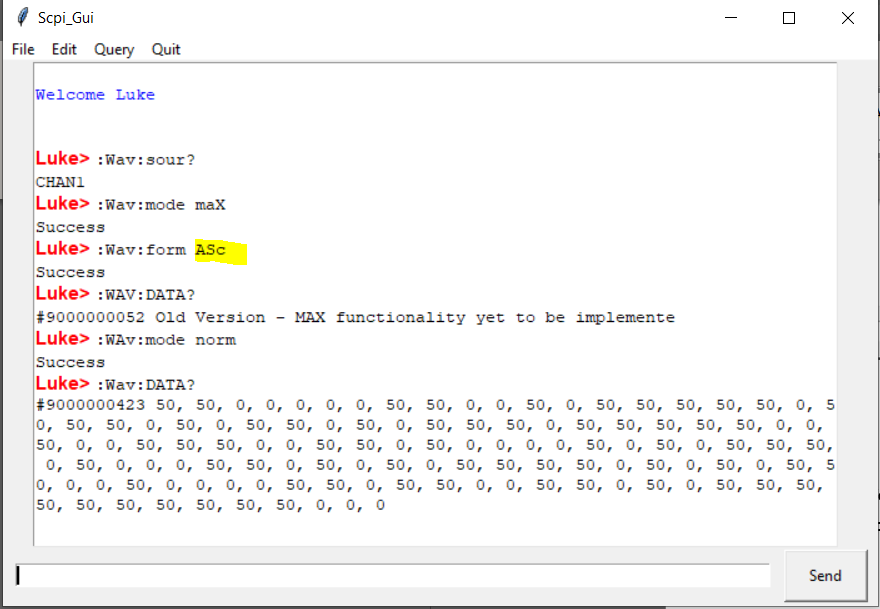




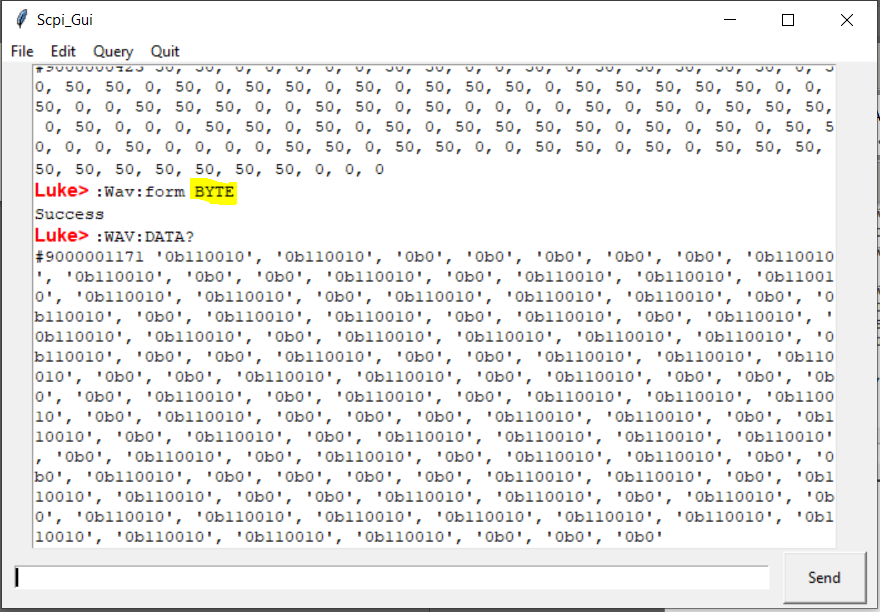
**6**

**The ability for the client to use the “BYTE”, “WORD” and “ASCII” forms of the “:WAVeform:DATA?” command to retrieve data from the stub-server**

ASCii 🡪



BYTE 🡪



Word 🡪



**7**

**Implement the following commands on the stub-server, with the server responding with a suitable dummy response**

\*CLS

\*RST

:WAVeform:XINCrement?

:WAVeform:XORigin?

:WAVeform:XREFerence?

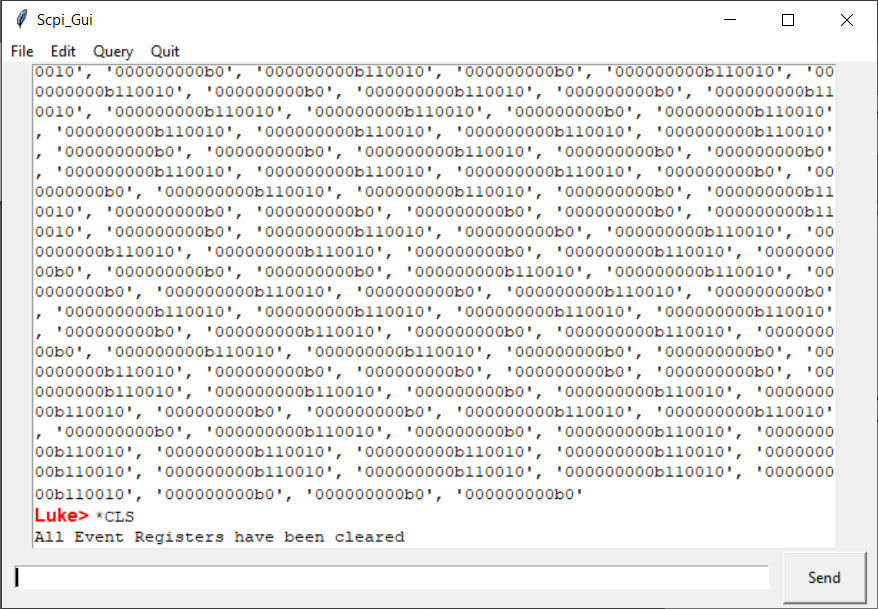
:WAVeform:YINCrement?

:WAVeform:YORigin?

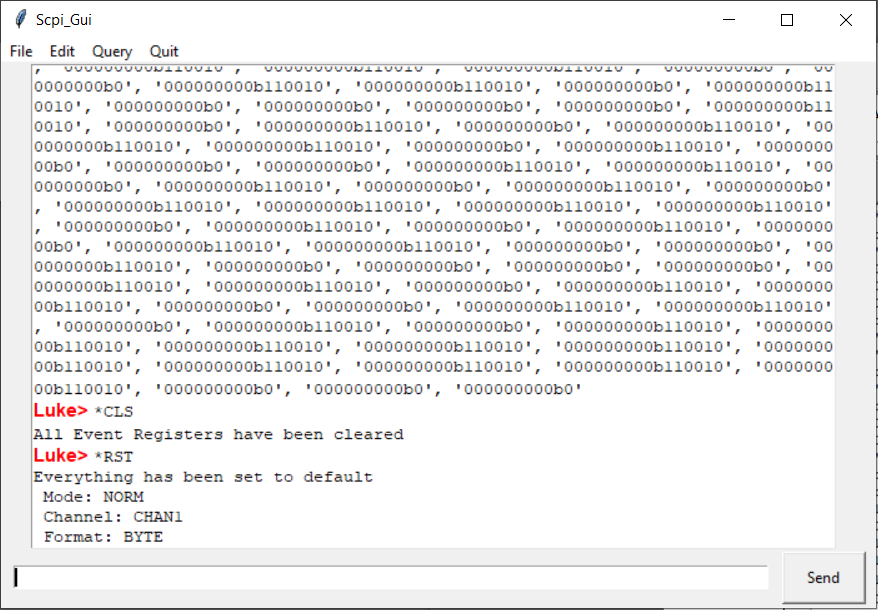
:WAVeform:YREFerence?

:WAVeform:PREamble?

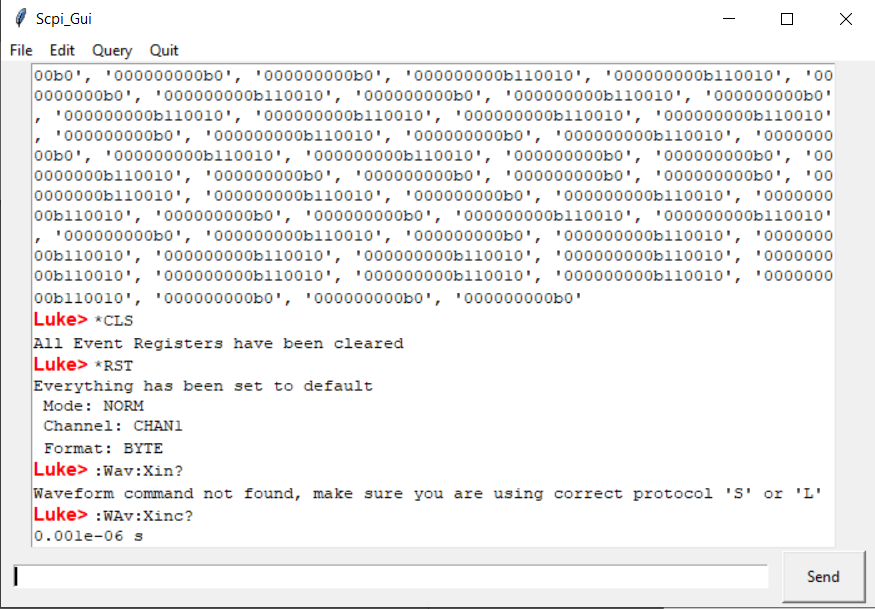
**7.1 \*CLS**



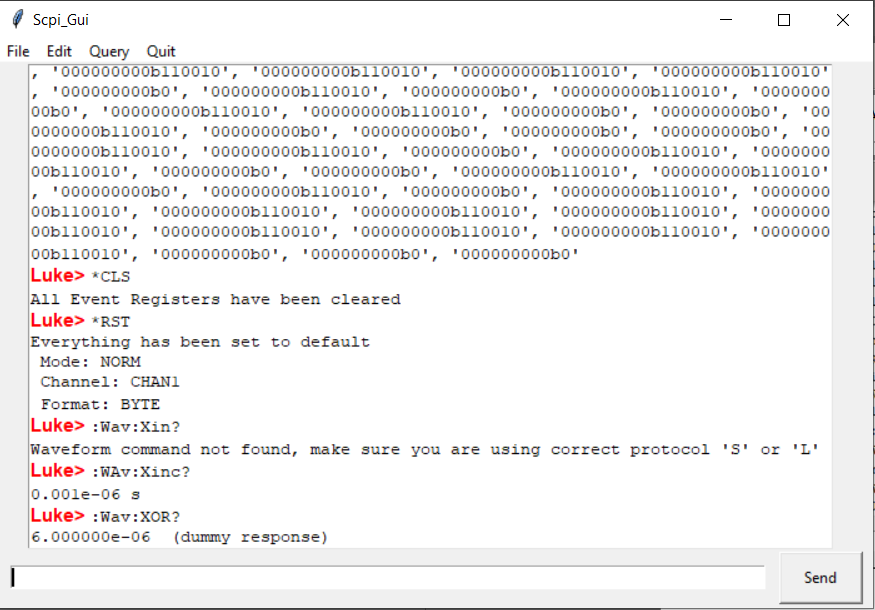
**7.2 \*RST**



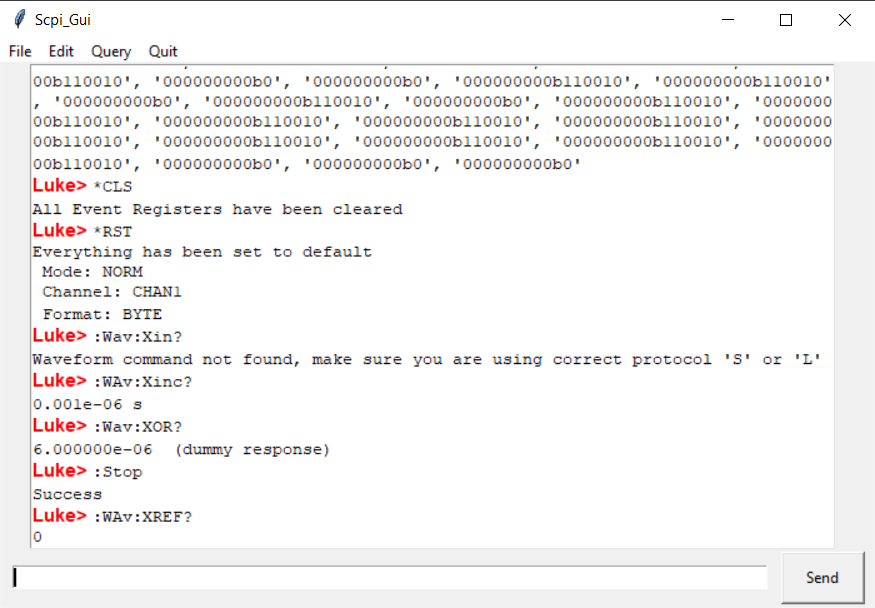
**7.3 :WAVeform:XINcrement?**



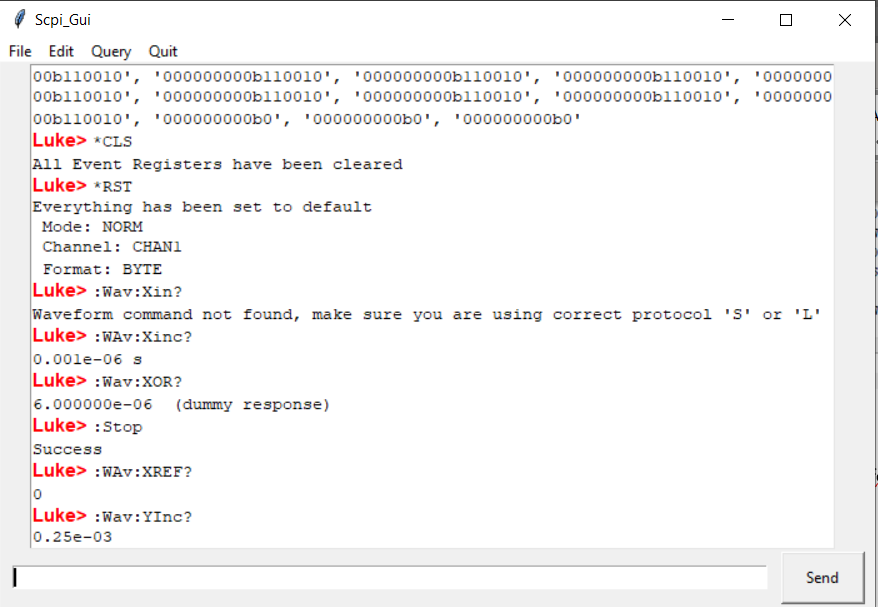
**7.4 :WAVeform:XORigin?**



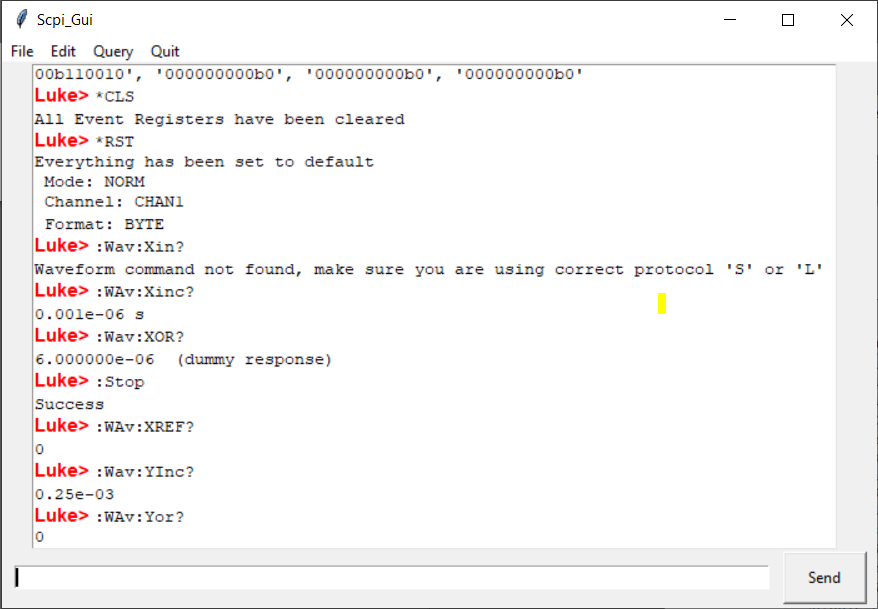
**7.5 :WAVeform:XREFerence?**



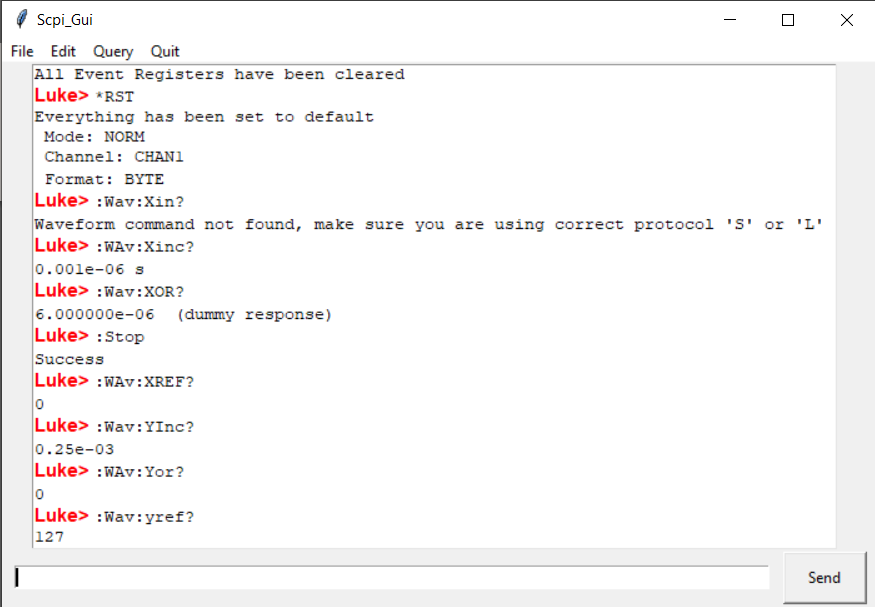
**7.6 :WAVeform:YINCrement?**



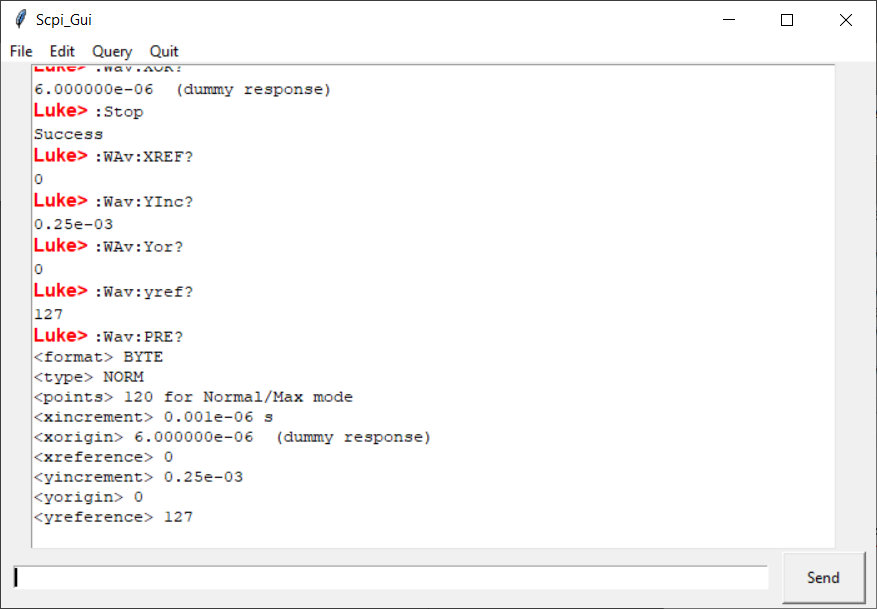
**7.7 :WAVeform:YOrigin?**



**7.8 :WAVeform:YREFerence?**

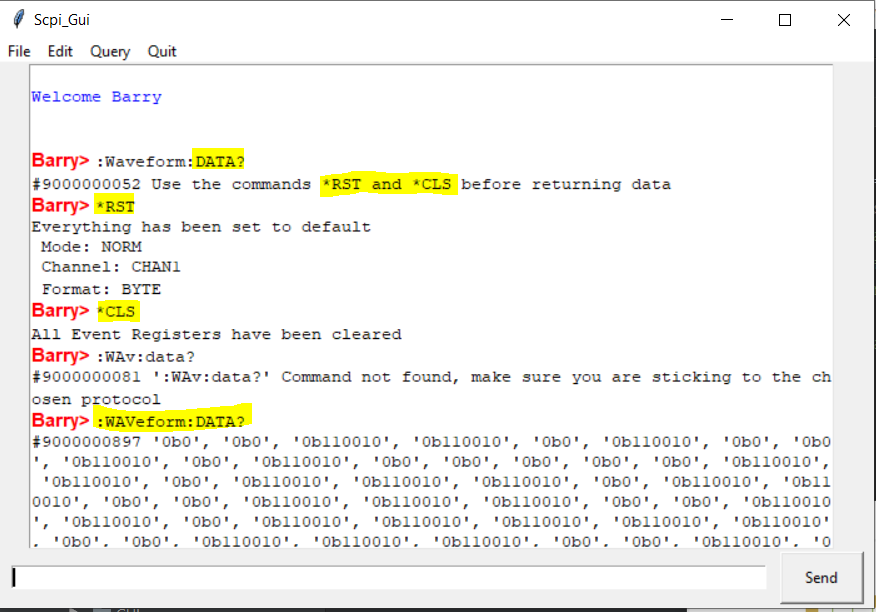


**7.9 :WAVeform:PREamble?**



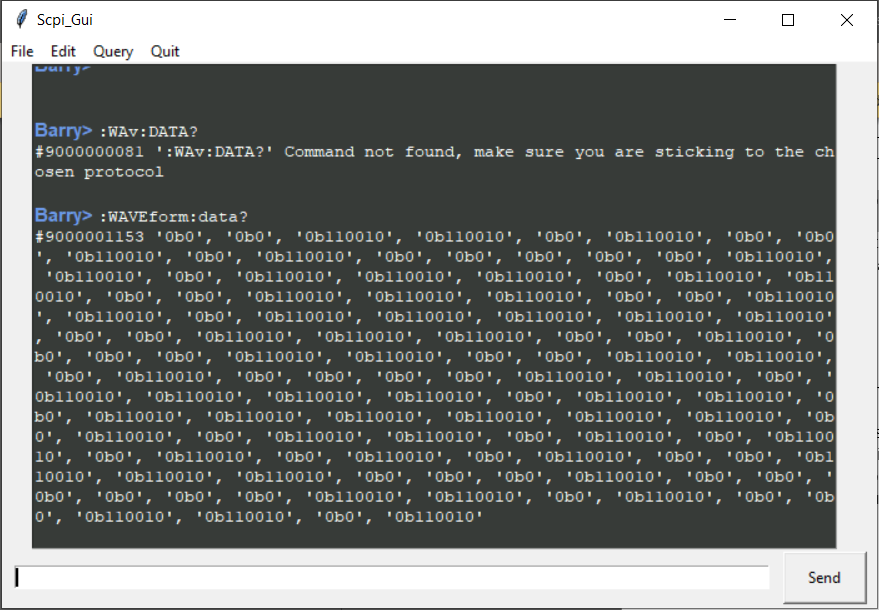
**8**

**The client should use the command sequence “\*RST” and “\*CLS” before attempting to return data.**



**9**

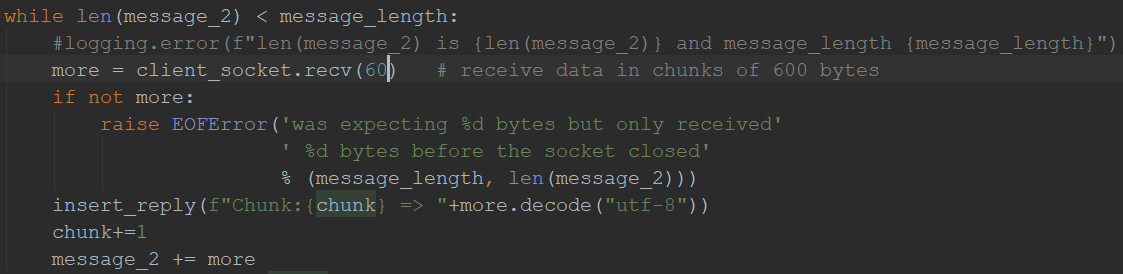
**A parsing routine in the client which will take the data returned by “:WAVeform:DATA?” and return it in a suitable data-structure, e.g. an Python function “getWaveData” returning a list of data items.**



**10**

**The ability to read the wave-form data from the stub-server in batches, instead of a single stream.**

Printing out chunks of data:

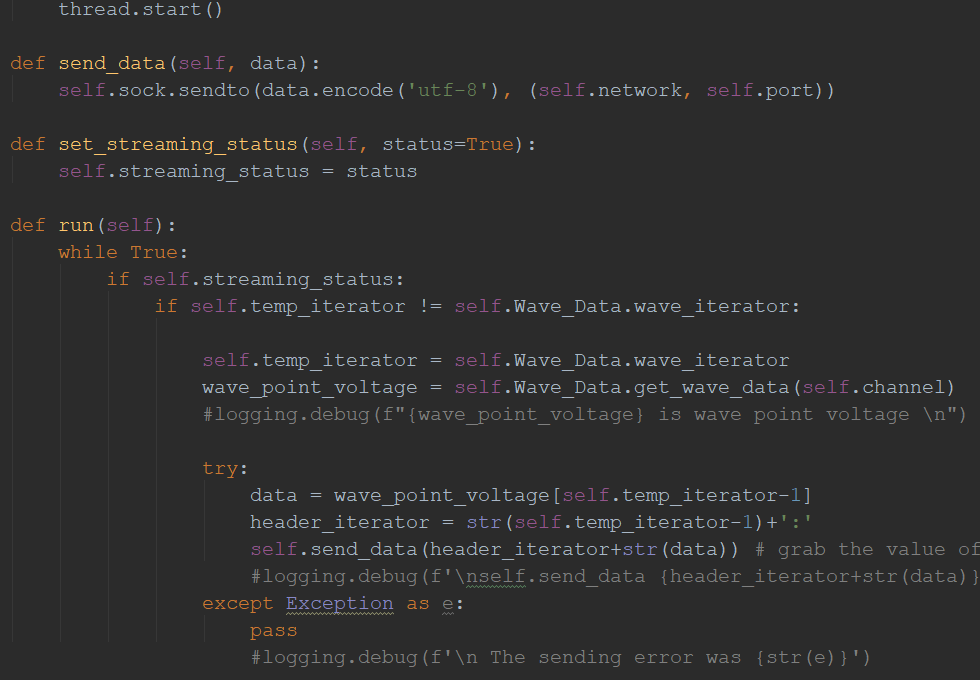
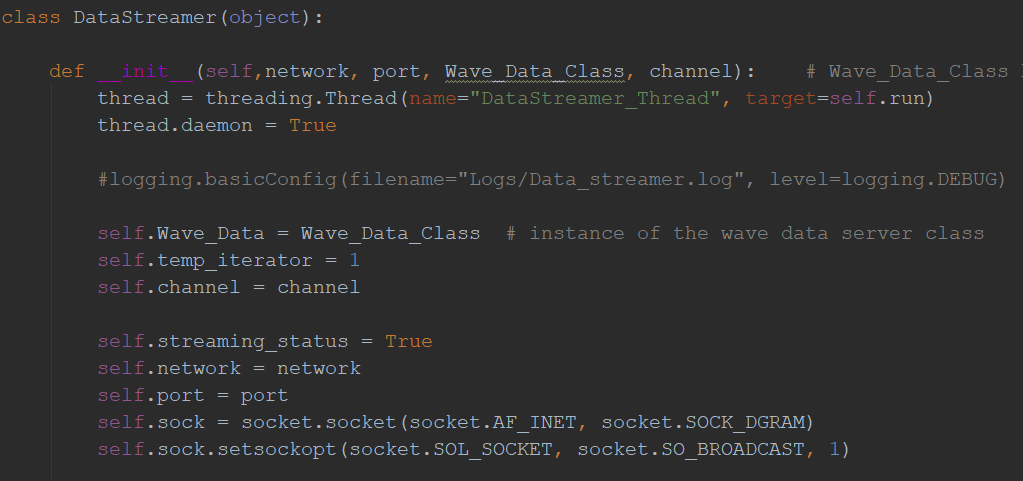




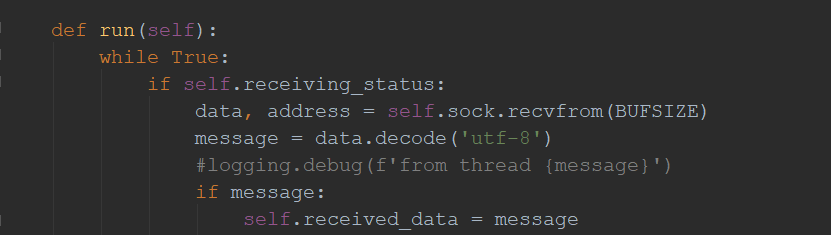
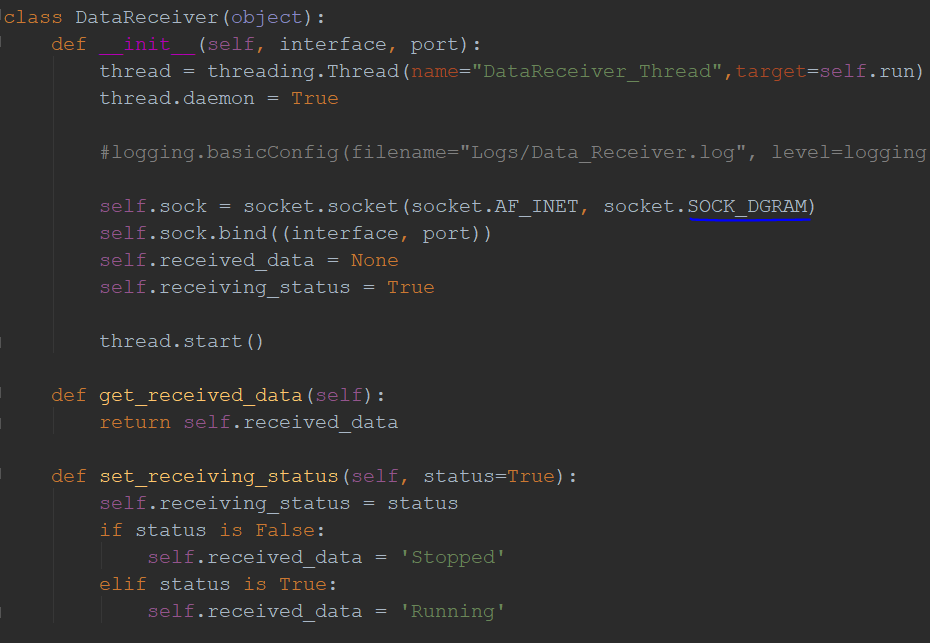
**11**

**An ability to use both the TCP/PI socket interface and another suitable bus to communicate between the loss-in-weight instrumentation and the drive control logic. NOTE: You do not have to implement the second communication method: just the abstraction in both the client and the stub-server that would be required to use to.**

Uses a UDP socket to live broadcast wave data to the client:



The client uses another UDP socket thread to receive the broadcasted data:



Allowing for the live streaming of the data:

