Neuro-1 API Guide

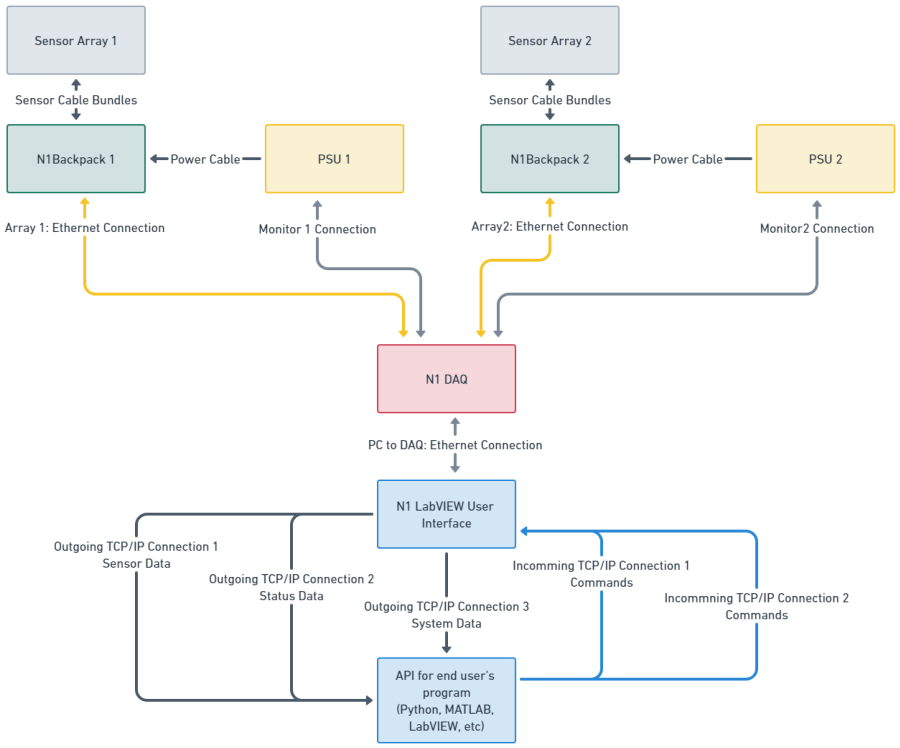
Introduction

Welcome to the Neuro-1 API Guide; a resource for developers creating custom software solutions for controlling a Neuro-1 system via TCP/IP connections. This document details the functionalities, protocols, and methods available in our API. Whether you're developing a simple monitoring tool or a complex control system, this guide will provide you with the necessary information to integrate your software seamlessly with our instruments.

Our API is built with flexibility and ease of use in mind, enabling you to send and receive data over TCP/IP connections. This guide will walk you through the setup process, demonstrate how to establish and manage connections, and explain how to send commands and receive responses from the instrument. By the end of this guide, you will have a thorough understanding of how to leverage our API for your custom software development needs, ensuring that you can maximize the potential of your instrument control solutions.

Overview of Data Flow

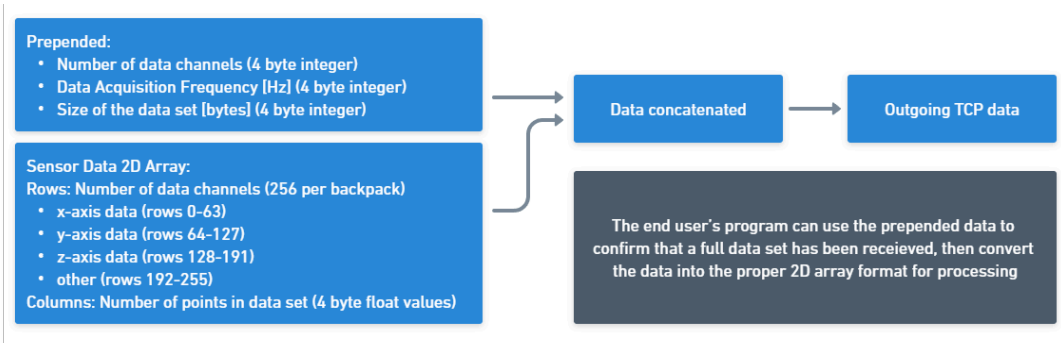
A basic data flow chart for the N-1 System is shown in Figure 1. Data from the sensor array is transmitted via Ethernet cable from the N-1 backpack, then to the digital acquisition box (DAQ) for processing, and finally to the user interface where it is displayed. External programs may monitor and control the N-1 system through the N-1 user interface using the three TCP/IP server ports for transmitting sensor data, sensor status, and system status level data to external programs, and the two TCP/IP command client ports. In the sections below, we review the data structure and provide example python code demonstrating how to access the N-1 system through these TCP/IP ports.



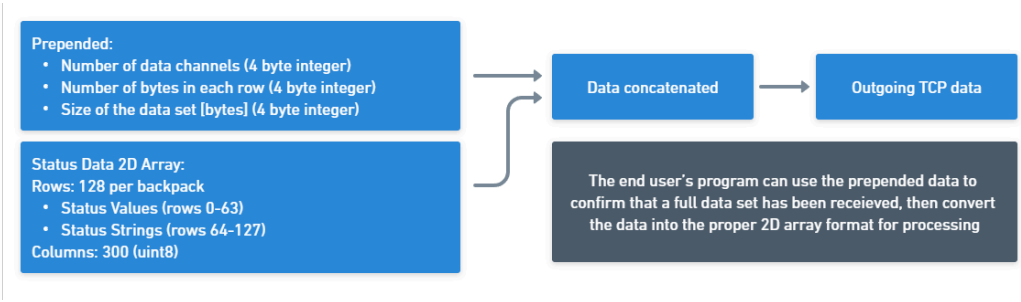
Monitoring N-1 Sensor and System Data

All data transmitted over the N-1 TCP/IP server ports are sent using a general format consisting of three 4 byte integers prepended to a larger data packet. The three prepended byte streams provide information about the incoming data packet, which the end user can use to reconstruct the data packet into a 2D array for further processing. Specific data flow charts showing the prepended data and the data packet are shown in Figures 2,3 and 4.

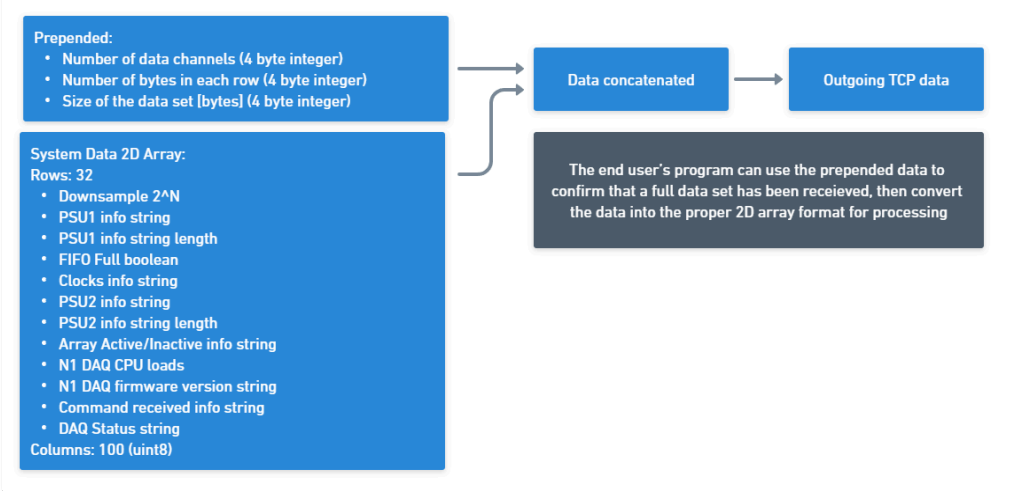
Sensor Data streamed from N-1 Server Port



Sensor Status Data streamed from N-1 Server Port



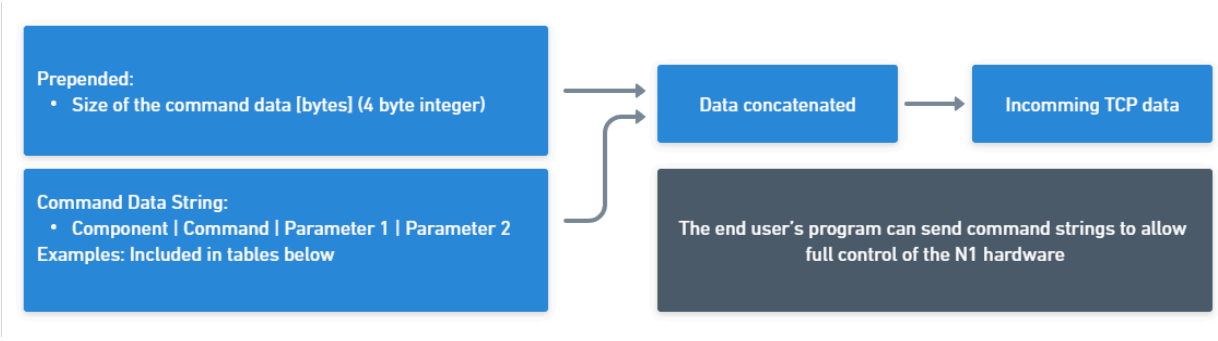
System Status Data streamed from N-1 Server Port



Controlling the N-1 System from the TCP/IP Command Port

Any control functionality provided by the N-1 user interface is also available to external users through the command client port. Commands are constructed using a the format: Component|Command|Parameter 1|Parameter 2, where component refers to either Sensor, DAQ, or PSU, and parameter refers to a number, or a range, depending on the command. Commands sent in this format sent over TCP/IP are received by the user interface, processed and then sent to their respective components. The general scheme is shown Figure 5 below. A complete list of commands, and their example usages are shown in Tables 1, 2, and 3 below.

Client Command Port Command Structure



Command Table with examples

Sensor Operations: Basic commands

Command	Description	Required Parameter	Optional Parameter	Example
Auto Start	Initiate sensor auto-start routine	N/A	N/A	Sensor|Auto Start
Reboot	Reboot the entire sensor array			Sensor|Reboot
Read Sensor Keys	Retrieve sensor key numbers			Sensor|Read Sensor Keys
Reset Field Coils	Erase field coil settings			Sensor|Reset Field Coils
Field Zero ON	Enable field zeroing routine			Sensor|Field Zero ON
Field Zero OFF	Disable field zeroing routine			Sensor|Field Zero OFF
Reset Ortho & Calibration Values	Erase all sensor orthogonalization and calibration values			Sensor|Reset Ortho & Calibration Values
Ortho & Calibrate	Engage orthogonalization and calibration routine			Sensor|Ortho & Calibrate
Closed Loop ON	Enable closed loop routine for all sensors			Sensor|Closed Loop ON
Closed Loop OFF	Disable closed loop routine for all sensors			Sensor|Closed Loop OFF
All ON (Change to Activate All)	Activate all sensors			Sensor|All ON (Change to Activate All)
All OFF (Change to Deactivate All)	Deactivate all sensors			Sensor|All OFF (Change to DeActivate All)
Update Channels	Assign sensors positions in array			Sensor|Update Channels
Module A ON/OFF	Turn on/off all sensors in module A	Array number: 1 or 2	0 = off, 1 = on, toggle if no parameter listed	Sensor|Module A ON/OFF|1|0
Module B ON/OFF	Turn on/off all sensors in module B			Sensor|Module B ON/OFF|2|1
Module C ON/OFF	Turn on/off all sensors in module C			Sensor|Module C ON/OFF|2|
Module D ON/OFF	Turn on/off all sensors in module D			Sensor|Module D ON/OFF|1|1
Module E ON/OFF	Turn on/off all sensors in module E			Sensor|Module E ON/OFF|2|0
Module F ON/OFF	Turn on/off all sensors in module F			Sensor|Module F ON/OFF|1|0
Module G ON/OFF	Turn on/off all sensors in module G			Sensor|Module G ON/OFF|1|0

Module H ON/OFF	Turn on/off all sensors in module H			Sensor|Module H ON/OFF|1|0
Active Sensors	Activate sensors specified by the range	Range of sensor numbers, 1-128 (use commas or dashes to separate)		Sensor|Active Sensors|2,6,7-55|1

Sensor Operations: Advanced commands (no required parameters)

Command	Description	Example
Cycle inc value	Change the increment size for Bx/By/Bz ++/-- commands	Sensor|Cycle inc value
Bx++	Increment field on Bx coil ++	Sensor|Bx++
Bx--	Increment field on Bx coil --	Sensor|Bx--
By++	Increment field on By coil ++	Sensor|By++
By--	Increment field on By coil --	Sensor|By--
Bz++	Increment field on Bz coil ++	Sensor|Bz++
Bz--	Increment field on Bz coil --	Sensor|Bz--
Bx MOD OFF	Turn off Bx coil modulation	Sensor|Bx MOD OFF
By MOD OFF	Turn off By coil modulation	Sensor|By MOD OFF
Bz MOD OFF	Turn off Bz coil modulation	Sensor|Bz MOD OFF
All B MOD OFF	Turn off Bx,By,Bz coil modulations	Sensor|All B MOD OFF
Bx MOD ON	Turn on Bx coil modulation	Sensor|Bx MOD ON
By MOD ON	Turn on By coil modulation	Sensor|By MOD ON
Bz MOD ON	Turn on Bz coil modulation	Sensor|Bz MOD ON
All B MOD ON	Turn on Bx,By,Bz coil modulations	Sensor|All B MOD ON
Only Ortho	Run orthogonalization procedure	Sensor|Only Ortho
Only Calibrate	Run calibration procedure	Sensor|Only Calibrate
Reset Ortho Values	Reset only the orthogonalization parameters	Sensor|Reset Ortho Values
Reset Calibration Values	Reset only calibration parameters	Sensor|Reset Calibration Values

DAQ and PSU Operations (no required parameters)

Command	Description	Example
Set Frequency: 1500 Hz	Sets the data sampling rate to 1500 Hz	DAQ|Set Frequency: 1500 Hz
Set Frequency: 750 Hz	Sets the data sampling rate to 750 Hz	DAQ|Set Frequency: 750 Hz
Set Frequency: 375Hz	Sets the data sampling rate to 375 Hz	DAQ|Set Frequency: 375 Hz
PSU Enable Outputs	Enables all power supplies	PSU|PSU Enable Outputs
PSU Disable Outputs	Disables all power supplies	PSU|PSU Disable Outputs

Example Control and Monitor Script: Python

The following Python script provides a basic implementation monitoring and controlling an N-1 system.

Python

```
import socket
import threading
import struct
import numpy as np
import time
import sys
from datetime import timedelta
import os

message = ""
status_string = ""
server_connection = 0
sensor_data_to_print = [10]
status_data_to_print = [300]
system_data_to_print = [100]
sensor_data_ready = 0
status_data_ready = 0
system_data_ready = 0

# Client-side communication
def client_connect(address):
  client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  client_socket.connect(address)
  return client_socket

def handle_client(client_socket, data_type):
  expected_size = 0
  while True:
    if expected_size == 0:
      # Attempt to receive the header in a robust way
      header = b''
            while len(header) < 12:
                part = client_socket.recv(12 - len(header))
                if not part:
                    raise ConnectionError("Connection lost while receiving the header")
                header += part

            rows, columns, expected_size = struct.unpack('iii', header)

    accumulated_data = b""
```

```
        while len(accumulated_data) < expected_size:
            try:
                # Receive the remaining data
                chunk = client_socket.recv(expected_size - len(accumulated_data))
                if not chunk:
                    raise ConnectionError("Connection lost during data reception")
        accumulated_data += chunk
      except Exception as e:
        print(f"Error: {e}")
                sys.exit(1)

        if len(accumulated_data) == expected_size:
            dtype = np.float32 if data_type == 'sensor' else np.uint8
            data = np.frombuffer(accumulated_data, dtype=dtype)

            # Process the received data here
            if data_type == 'sensor':
                global sensor_data_to_print
                sensor_data_to_print[0] = data[0]
                global sensor_data_ready
                sensor_data_ready = 1
            elif data_type == 'status':
                global status_data_to_print
                status_data_to_print = data[0:300]
                global status_data_ready
                status_data_ready = 1
            elif data_type == 'system':
                global system_data_to_print
                system_data_to_print = data[0:100]
                global system_data_ready
                system_data_ready = 1
        else:
            print(f"Data size mismatch: expected {expected_size}, got {len(accumulated_data)}")
            sys.exit(1)

        # Reset expected_size for the next cycle
        expected_size = 0

def process_data(data_array):
    try:
        data_list = data_array.tolist()  # Convert Numpy array to a list
        end_index = data_list.index(13) + 1 if 13 in data_list else len(data_list)
        return bytes(data_list[:end_index]).decode('ascii', errors='ignore')
```

```
    except ValueError:
        print("Error processing data.")
        return ""

# Elapsed time printing modified to update on the same line
def print_elapsed_time():
    start_time = time.time()
    while True:
        global message
        global server_connection
        global sensor_data_to_print
        global status_data_to_print
        global system_data_to_print

        elapsed_time = time.time() - start_time
        delta = timedelta(seconds=int(elapsed_time))
        if(server_connection == 1 and sensor_data_ready == 1 and status_data_ready == 1 and
system_data_ready == 1):
            # Process and decode status_data_to_print
            status_string = process_data(status_data_to_print)

            # Process and decode system_data_to_print
            system_string = process_data(system_data_to_print)

            os.system('cls')
            print(f"Elapsed Time: {delta}")
            print(f"Sensor Data: {sensor_data_to_print[0]}")
            print(f"Status Data: {status_string}")
            print(f"System Data: {system_string}")
            print(f"Message to client: {message}")
        time.sleep(0.2)

# Server communication function modified for inline input
def server_communication():
    SERVER_IP = 'localhost'
    SERVER_PORT = 8092
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((SERVER_IP, SERVER_PORT))
    server_socket.listen()
    print(f"Server listening on {SERVER_IP}:{SERVER_PORT}")

    client_socket, client_address = server_socket.accept()
    print(f"Client connected from {client_address}")
```

```
    global server_connection
    server_connection = 1

    while True:
        global message
        message = input()
        if message == 'exit':
            break

        message_bytes = message.encode()
        header = struct.pack('!I', len(message_bytes))
        client_socket.sendall(header + message_bytes)

    client_socket.close()
    server_socket.close()

# Main script execution
if __name__ == "__main__":
  client_socket1 = client_connect(('localhost', 8089))
  client_socket2 = client_connect(('localhost', 8090))
  client_socket3 = client_connect(('localhost', 8091))

  thread1 = threading.Thread(target=handle_client, args=(client_socket1, 'sensor'))
  thread2 = threading.Thread(target=handle_client, args=(client_socket2, 'status'))
  thread3 = threading.Thread(target=handle_client, args=(client_socket3, 'system'))
  thread4 = threading.Thread(target=server_communication)
  thread5 = threading.Thread(target=print_elapsed_time)

  thread1.start()
  thread2.start()
  thread3.start()
  thread4.start()
  thread5.start()
```