

ThinkGear Socket Protocol

Introduction

The ThinkGear Socket Protocol (TGSP) is a [JSON-based](#) protocol for the transmission and receipt of ThinkGear brainwave data between a client and a server. TGSP was designed to allow languages and/or frameworks without a standard serial port API (e.g. Flash and most scripting languages) to easily integrate brainwave-sensing functionality through socket APIs.

This document is a specification for TGSP and can be used in conjunction with the ThinkGear Connector applications on the PC and Mac.

Conventions

There will be several nomenclature conventions that will be used throughout this document.

- A **server** is a device or application that implements TGSP, and is responsible, amongst other things, for responding to authorization requests and broadcasting headset data. The ThinkGear Connector is an example of a “server”.
- A **client** is a device or application that connects to a server.
- **Headset data** refers to the data returned by a headset containing a ThinkGear module.

JSON nomenclature conventions will also be used throughout this document (primarily the concept of a JSON *object*), so it is best to [scan the language specification](#) to brush up.

Overview

There are several primary stages in the lifetime of a TGSP connection.

1. **Creation of socket connection**
2. **Authorization** (one-time) – Authorization of the client by the server
3. **Configuration of server** (performed any time)
4. **Receipt of headset data** (repeating)
5. **Termination of socket connection**

These primary stages are covered in detail in the following sections.

Authorization

The client must initiate an authorization request and the server must authorize the client before the server will start transmitting any headset data.

Parameters

- **appName. Required.** A human-readable name identifying the client application. This can be a maximum of 255 characters.
- **appKey. Required.** The key used by the client application to identify itself. This must be 40 hexadecimal characters, ideally generated using an SHA-1 digest. See the Note below.

```
{"appName": "Brainwave Shooters", "appKey": "9f54141b4b4c567c558d3a76cb8d715cbde03096"}
```

The appKey is an identifier that is unique to each application, rather than each *instance* of an application. It is used by the server to bypass the authorization process if a user had previously authorized the requesting client. To reduce the chance of overlap with the appKey of other applications, the appKey should be generated using an SHA-1 digest.

Response

The server will respond to the client after receiving an authorization request from the client. The response will be sent prior to the transmission of any headset data.

- **isAuthorized.** Tells the client whether the server has authorized access to the user's headset data. The value is either `true` or `false`.

```
{"isAuthorized": true}
```

There is **no guarantee** that a response to the authorization request will be transmitted by the server in any amount of time. As such, clients should stay in an idle state until a response is received from the server.

Configuration

A client can send commands to a server to configure such things as transmission formats or the components of data transmitted by the server. These commands can be sent at any time after the authorization process.

Parameters

TODO DEPRECATED

- `enableRawOutput` - *Optional*. Whether raw sensor output should be included in the transmitted data. The value of this parameter should be either `true` or `false` (default).
- `format` - *Optional*. The format in which headset data should be transmitted to the client. The value of this parameter should be either `"BinaryPacket"` (default) or `"Json"`. When specifying this value, **make note of the capitalization!**

```
{"enableRawOutput":true,"format":"Json"}
```

Response

TODO DEPRECATED No explicit response to these packets will be sent by the server – the server will simply start transmitting data in the configured format.

Because it may take some time for the ThinkGear Connector to re-configure itself to transmit JSON packets, several binary packets may be prematurely transmitted to the application. As such, an application should be able to handle the receipt of unexpected binary packets without failing critically.

Update: The Binary Socket Packet Format is now deprecated in favor of the JSON format.

Headset Data Transmission

TODO DEPRECATED Data transmission from the server is done using a streaming model; the client does not issue any explicit requests to the server for brainwave data.

Because there is no mechanism in JSON to handle streaming (i.e. continuously appended) data, TGSP delimits individual JSON objects with carriage return characters (`\r`), so each JSON object will occupy its own line.

TODO: Deprecate

Even though JSON is the preferred transmission format, the **binary packet format** (used in earlier

versions of TGSP) is the **default** format. Documentation for the binary packet format can be found in the [Binary Socket Packet Format](#) document.

The Binary Socket Packet Format will **eventually be deprecated** in favor of the JSON format, so application developers are encouraged to switch to the JSON format as soon as possible.

Update: The Binary Socket Packet Format is now deprecated in favor of the JSON format.

Response

- **poorSignalLevel**. A quantifier of the quality of the brainwave signal. This is an integer value that is generally in the range of 0 to 200, with 0 indicating a good signal and 200 indicating an off-head state.
- **eSense**. A container for the eSense™ attributes. These are integer values between 0 and 100, where 0 is perceived as a lack of that attribute and 100 is an excess of that attribute.
 - **attention**. The eSense Attention value.
 - **meditation**. The eSense Meditation value.
- **eegPower**. A container for the EEG powers. These may be either integer or floating-point values.
 - **delta**. The “delta” band of EEG.
 - **theta**. The “theta” band of EEG.
 - **lowAlpha**. The “low alpha” band of EEG.
 - **highAlpha**. The “high alpha” band of EEG.
 - **lowBeta**. The “low beta” band of EEG.
 - **highBeta**. The “high beta” band of EEG.
 - **lowGamma**. The “low gamma” band of EEG.
 - **highGamma**. The “high gamma” band of EEG.
- **rawEeg**. The raw data reading off the forehead sensor. This may be either an integer or a floating-point value. This data is represented in its own JSON object, as in the sample below.
- **rawEegMulti**. A container for multichannel raw EEG data. These may be either integer or floating-point values.
 - **ch1**. The raw data from channel 1.
 - **ch2**. The raw data from channel 2.
 - **ch3**. The raw data from channel 3.
 - **ch4**. The raw data from channel 4.
 - **ch5**. The raw data from channel 5.
 - **ch6**. The raw data from channel 6.
 - **ch7**. The raw data from channel 7.
 - **ch8**. The raw data from channel 8.
- **blinkStrength**. The strength of a detected blink. This is an integer in the range of 0-255. This data is represented in its own JSON object, as in the sample below.
- **mentalEffort(BETA)**. The mentalEffort of doing a task. This is a decimal number. This data is represented in its own JSON object, as in the sample below.
- **familiarity(BETA)**. The familiarity of doing a task. This is a decimal number. This data is represented in its own JSON object, as in the sample below.

```
{ "poorSignalLevel": 0, "eSense": { "attention": 38, "meditation": 43 }, "eegPower": { "delta": 1.15e-4, "theta": 1.41e-6, "lowAlpha": 1.35e-4, "highAlpha": 6.69e-5, "lowBeta": 1.47e-5, "highBeta": 6.95e-7, "lowGamma": 5.26e-7, "highGamma": 1.40e-5 } }
```

```
{"rawEeg":238}
{"rawEeg":282}
{"blinkStrength":100}
{"mentalEffort":2.17604843163533}
{"familiarity":391.789551397294}
{"rawEeg":239}
{"rawEegMulti":{"ch1":392,"ch2":352,"ch3":492,"ch4":592,"ch5":692,"ch6":442,
"ch7":122,"ch8":552}}
```

With the exception of `rawEeg` and `blinkStrength`, the headset components are transmitted at a rate of 1Hz. `rawEeg`, if enabled, is transmitted at a rate no higher than 512Hz. `blinkStrength` is transmitted whenever a blink is detected by the headset.

The client should **not** expect a specific component of headset data to be present in all (or even any) packets transmitted by the server. The client should thus maintain state between receipts of headset data from the server. Also, the ordering of the parameters in each individual JSON object cannot be guaranteed.

Recording

If the server supports brainwave and event recording, the client can enable recording by sending the following commands:

Start recording

The client can start recording by sending:

```
{"startRecording":{"rawEeg":true,"poorSignalLevel":true,"eSense":true,
"eegPower":true,"blinkStrength":true},"applicationName":"ExampleApp"}
```

Parameters

- **startRecording**. **Required**. Container for parameters to enable recording types.
 - `rawEeg`. *Optional*. Set as `true` to enable raw EEG recording. Omit or set as `false` to disable raw EEG recording.
 - `poorSignalLevel`. *Optional*. Set as `true` to enable `poorSignalLevel` recording. Omit or set as `false` to disable `poorSignalLevel` recording.
 - `eSense`. *Optional*. Set as `true` to enable `eSense` recording. Omit or set as `false` to disable

- eSense recording.
- eegPower. *Optional*. Set as true to enable EEG power recording. Omit or set as false to disable EEG power recording.
- blinkStrength. *Optional*. Set as true to enable blink recording. Omit or set as false to disable blink recording.
- applicationName. **Required**. A human-readable name identifying the client application.

Response

None.

Stop recording

The client can stop the recording by sending:

```
{"stopRecording": "ExampleApp"}
```

Parameters

- stopRecording - **Required**. Set as the application name.

Response

The server will respond with the following:

```
{"status": "recordingStopped", "sessionId": 1234, "filePath": "c:\\path\\to\\file\\1234.json"}
```

- sessionId. The session number that was just recorded.
- filePath. The path to the recorded data.

Cancel recording

The client can cancel the recording by sending:

```
{"cancelRecording": "ExampleApp"}
```

Parameters

- `cancelRecording` - **Required**. Set as the application name.

Response

The server will respond with the following:

```
{"status": "canceled"}
```

Event Recording

During the recording, the client can record events by sending:

```
{"eventType": "question", "eventData": {"question": "question",  
  "answer": "answer", "result": true},  
  "time": 456123153, "applicationName": "ExampleApp"}
```

or

```
{"eventType": "generic", "eventData": {"genericJSONObject"}, "time": 45312255,  
  "applicationName": "ExampleApp"}
```

Parameters

- `eventType` - **Required**. Type of event. Either “question” or “generic”.
- `eventData` - **Required**. Container of event data.
 - For question - event types, the following parameters are **required**.
 - `question` - **Required**. Questioned asked by application.
 - `answer` - **Required**. Correct answer.
 - `userAnswer` - **Required**. Answer given by user.
 - `result` - **Required**. True or false depending on the user answering the question correctly.
 - For generic event types, `eventData` can hold any JSON object.
- `time` - *Optional*. Timestamp recorded by the application. (The server will automatically record a timestamp).
- `applicationName` - **Required**. Application name.

Response

None.

sessionId Retrieval

Retrieve previously recorded session ids from the server.

```
{"getSessionIds":"ExampleApp"}
```

Parameters

- getSessionIds - **Required.** Set as the application name.

Response

The server will respond with an array of sessionIds and the time the session started.

```
{"availableSessionIds":[{"sessionId":1,"timeStamp":198328888.222}, {"sessionId":2,"timeStamp":2828828.333}]}
```

If no sessions are available, the server will respond a null packet.

```
{"availableSessionIds":null}
```

Session Retrieval

The client can get a session's data by sending:

```
{"getSessionId":1,"applicationName":"ExampleApp"}
```

Parameters

- getSessionId - **Required.** The desired sessionId.
- applicationName - **Required.** Set as the application name.

Response

The server will respond with the desired session data


```
{"sessionId":1,"data":[{"timeStamp":3929239,
"poorSignalLevel":0,"eSense":{"attention":38,"meditation":43},
"eegPower":{"delta":1.15e-4,"theta":1.41e-6,"lowAlpha":1.35e-4,
"highAlpha":6.69e-5,"lowBeta":1.47e-5,"highBeta":6.95e-7,
"lowGamma":5.26e-7,"highGamma":1.40e-5}}, {"timeStamp":3929242,
"rawEeg":344}, {"timeStamp":3929245,"rawEeg":804}]}
```

Due to the large amounts of data returned, session data will be sent to only to clients in EventListener mode.

Parameters

- `sessionId` - The sessionId.
- `data` - An array of timestamped session data.

Event listening

To get a live stream of events from another application, send the following command:

```
{"enableRawOutput":true,"format":"EventListener"}
```

Parameters

- `enableRawOutput` - *Optional*. Whether raw sensor output should be included in the transmitted data. The value of this parameter should be either `true` or `false` (default).
- `format` - **Required**. Set as `EventListener`.

Response

None.

List Applications

The client can request a list of applications with recorded data by sending:

```
{"getAppNames":null}
```

Parameters

- getAppNames - **Required.** Set as null.

Response

The server will respond with a list of applications names:

```
{"appNames": [ExampleApp, ExampleApp2]}
```

If there are no recorded data, then the server will respond:

```
{"appNames": null}
```

Parameters

- appNames - An array of application names.

Setting Users

To enable recording by user, send the following command:

```
{"setUser":{"userName":"Joe Smith", "userId":12}}
```

Parameters

- userName - **Required.** Set as desired username.
- userId - *Optional.* Set as desired ID number. If not supplied, the server will automatically assign the next available ID.

Response

If successful, the server will respond with:

```
{"setUserSuccess":{"userName":"Joe Smith", "userId":12}}
```

Getting Users

To enable recording by user, send the following command:

```
{"getUsers": "exampleApp"}
```

Parameters

None.

Response

If successful, the server will respond with:

```
{"users": [{"userName": "Joe Smith", "userId": 2}, {"userName": "Matt Yoon", "userId": 4}]}
```

Deleting Users

To enable recording by user, send the following command:

```
{"deleteUser": {"userName": "Joe Smith", "userId": 2}}
```

Parameters

- **userName** - **Required**. Set as desired username.
- **userId** - **Required**. Set as desired ID number.

Response

If successful, the server will respond with:

```
{"deleteUserSuccess": {"userName": "Joe Smith", "userId": 12}}
```

Parsing

Clients will first have to tokenize the stream using the carriage return (\r) delimiter, then parse each token individually as a JSON object. This is demonstrated by the following pseudocode:

```
while there is still data in the stream
    read the line
    parse the line as JSON
```

When using the JSON output format and tokenizing a packet stream using a \r delimiter, be careful about parsing the last token as a JSON object. The packet stream will end in a \r character, meaning that the tokenizer will likely return an empty string as the last token.

Also, your parsing code should be tolerant of incomplete packet strings, in the event that the stream is parsed mid-transfer.

Once a JSON object has been extracted out of the stream, it can be parsed using any of a number of readily-available JSON parsing libraries. An exhaustive list of JSON parsers for various languages can be found at the [JSON website](#), but here are the ones that NeuroSky recommends:

Language	Library	
ActionScript 3 (Flash/Flex)	https://github.com/mikechambers/as3corelib	
C# (.NET/Mono)	https://code.google.com/p/jayrock/	Jayrock

ActionScript 3 (Adobe Flash and Flex)

Once a Socket has been created in your code, you'll need to configure the ThinkGear Connector to output JSON (and optionally, raw sensor data). This is done by sending a packet that is formatted to the [Configuration packet specification](#). For example:

```
var configuration : Object = new Object();
configuration["enableRawOutput"] = true;
configuration["format"] = "Json";

socket.writeUTFBytes(JSON.encode(configuration));
```

When reading data from the ThinkGear Connector, you can read data directly into a String from the socket stream. For AS3, this code would typically go into the function that was delegated as the event listener:

```
var packetString : String = socket.readUTFBytes(socket.bytesAvailable);
```

Then, the string should be tokenized using the carriage return (\r) as the delimiter:

```
var packets : Array = packetString.split(/\r/);
```

You can then iterate over each of the packets, parsing it into JSON:

```
for(var packet : String in packets){  
    var data : Object = JSON.decode(packet);  
  
    // note that not all packets will contain a "rawEeg" parameter; the  
    // appropriate error checking should be performed.  
    trace(data["rawEeg"]);  
    trace(data["eSense"]["attention"]);  
}
```

C# (.NET and Mono)

Typically, socket data is returned as an array of bytes in a buffer. This should be converted to a string prior to parsing it as JSON:

```
byte[] buffer = new byte[8192];  
networkStream.Read(buffer, 0, buffer.Length);  
  
string packetString = System.Text.ASCIIEncoding.ASCII.GetString(buffer);
```

Next, the string should be tokenized using a carriage return (\r) as the delimiter:

```
string[] packets = String.Split(packetString, new char[]{"\r"});
```

Now that you've split the packet stream into its constituent packets, you can loop over the array and parse each packet individually. The headset data can then be referenced directly:

```
foreach(string packet in packets){  
    IDictionary data = (IDictionary)JsonConvert.Import(typeof(IDictionary),  
    packet);  
  
    // note that not all packets will contain a "rawEeg" parameter; the  
    // appropriate error checking should be performed.  
    Console.WriteLine("Raw data: " + data["rawEeg"]);  
}
```

By default, Visual Studio imports the `System.Collections.Generic` package when creating a new class file. During compilation, however, this causes problems with the typecast used above. Simply remove the `import System.Collections.Generic` statement from the file header to fix the compilation error.

From:

<http://developer.neurosky.com/docs/> - **NeuroSky Developer - Docs**

Permanent link:

http://developer.neurosky.com/docs/doku.php?id=thinkgear_socket_protocol

Last update: **2014/06/18 22:16**



Warnings and Disclaimer of Liability

THE ALGORITHMS MUST NOT BE USED FOR ANY ILLEGAL USE, OR AS COMPONENTS IN LIFE SUPPORT OR SAFETY DEVICES OR SYSTEMS, OR MILITARY OR NUCLEAR APPLICATIONS, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE ALGORITHMS COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR. YOUR USE OF THE SOFTWARE DEVELOPMENT KIT, THE ALGORITHMS AND ANY OTHER NEUROSKY PRODUCTS OR SERVICES IS "AS-IS," AND NEUROSKY DOES NOT MAKE, AND HEREBY DISCLAIMS, ANY AND ALL OTHER EXPRESS AND IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, AND ANY WARRANTIES ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL NEUROSKY BE LIABLE FOR ANY SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING BUT NOT LIMITED TO LOSS OF PROFITS OR INCOME, WHETHER OR NOT NEUROSKY HAD KNOWLEDGE, THAT SUCH DAMAGES MIGHT BE INCURRED.