

# ThinkGear SDK for .NET: Development Guide and API Reference

## Introduction

This guide will teach you how to use the **ThinkGear SDK for .NET** to write Windows apps that can utilize bio-signal data from NeuroSky's bio-sensors. This will enable your Windows apps to receive and use bio-signal data such as EEG acquired from NeuroSky's sensor hardware.

This guide (and the entire **ThinkGear SDK for .NET**) is intended for programmers who are already familiar with standard .NET development using Microsoft Visual Studio. If you are not already familiar with developing for .NET, please first visit <http://www.microsoft.com/net> to learn how to properly set up your .NET development environment and create typical .NET apps.

If you are already familiar with creating typical .NET apps, then the next step is to make sure you have downloaded the **ThinkGear SDK for .NET**. Chances are, if you're reading this document, then you already have it and can access the full folder specific to the ThinkGear SDK for .NET.

## ThinkGear SDK for .NET Contents

- **ThinkGear SDK for .NET: Development Guide and API Reference** (this document)
- **libs/**:
  - **ThinkGear.dll** library
  - **JayrockJson.dll** supporting library
  - **NLog.dll** supporting library
  - **NLog.xml** and **NLog.config** configuration files
  - **NLog.LICENSE.txt** and **Jayrock.LICENSE.txt** licence files
- **TG-HelloEEG.exe** - a reference build of the HelloEEG sample project
- **HelloEEG Sample Project** source code

You will find the .dll, configuration files and 3rd party license documents in the **libs/** folder. Copy this entire folder into your project.

You will find the source code of the "HelloEEG Sample Project" in the **Sample Projects/HelloEEG** folder.

## Supported NeuroSky Hardware

The ThinkGear SDK for .NET can only be used with the following NeuroSky devices, chips, and modules:

- MindWave Mobile
- MindWave (RF)
- MindBand
- ThinkCap
- TGAM module
- TGAT ASIC

Before using any Windows app that uses the ThinkGear SDK for .NET, make sure you have paired the sensor hardware device to your Windows machine by carefully following the instructions in the User Manual that came with each sensor hardware device! The sensor hardware device must properly appear in your Windows machine's list of COM ports in Device Manager.

## Your First Project: HelloEEG console

HelloEEG is a sample project we've included in the **ThinkGear SDK for .NET** that demonstrates how to setup, connect, and handle data to a NeuroSky device. Add the project to your Visual Studio by following these steps:

For Visual Studio:

1. from the Visual Studio Toolbar, select **File** -> **New** -> **Project From Existing Code...**
2. In the New Project From Existing Code wizard, select the project type of "Visual C#"
3. click the "Next >" button
4. browse to the place you have expanded the SDK files. ("ThinkGear SDK for .NET\Sample Projects\HelloEEG")
5. check the box to include subfolders.
6. enter a name of "HelloEEG"
7. choose Output type of "Console Application"
8. click the "Finish" button
9. at the Toolbar select **Project** -> **HelloEEG Properties...**
10. change the Assembly name to HelloEEG
11. set the Target framework to ".NET Framework 3.5"
12. if you are asked to Confirm the Framework change, click "Yes"
13. at the Toolbar select **View** -> **Solution Explorer**
14. in the Solution Explorer pane select and expand the "References" section
15. if you see a exclamation mark warning on "Microsoft.CSharp", right click on the warning, and "Remove" the reference to "Microsoft.CSharp"
16. select the "References" section, right click, pick "Add Reference.."
17. click the browse button, navigate to where you have expanded the SDK files, choose the folder "neurosky" and then pick "ThinkGear.dll"
18. at the Toolbar select **Build** -> **Build Solution**
19. if there are no errors, you should be able to browse the code, make modifications, compile, and run the app just like any typical .NET app.

For Visual Studio Express:

1. From the Visual Studio Express Toolbar, select **File** -> **Open Project**
2. Navigate to where you have expanded the SDK files, and then to *ThinkGear SDK for .NET\Sample Projects\HelloEEG\*
3. Select "HelloEEG.csproj" and click "Open"
4. From the Toolbar, select **Build** -> **Build Solution**
5. if there are no errors, you should be able to browse the code, make modifications, compile, and run the app just like any typical .NET app.

These steps have been tested with Visual Studio and Visual Studio Express 2010/2013, if yours is different you may have to adapt these instructions.

The TG-HelloEEG.exe reference program is built from these same sources and with the same process. It is slightly different in that the Microsoft ILMerge program has been used to incorporate the dlls from the /neurosky folder into the .exe so that it can function in a more standalone way.

# Developing Your Own NeuroSky Sensor Apps for .NET

## Preparing Your .NET Project

The **ThinkGear SDK for .NET**'s API is made available to your application via the `NeuroSky.ThinkGear` namespace. The **ThinkGear.dll** gives your .NET application access to the `NeuroSky.ThinkGear` namespace.

## The ThinkGear.dll

To start with, add the **ThinkGear.dll** file to your .NET application's project workspace. The **ThinkGear.dll** is a C# .NET library, and can only be used as part of .NET projects (it will not work in native projects). This .dll contains the `NeuroSky.ThinkGear` namespace.

## The NeuroSky.ThinkGear Namespace

The **ThinkGear SDK for .NET**'s API is made available to your application via the

NeuroSky.ThinkGear namespace. Once you have added the **ThinkGear.dll** file to your project, you can then add the following code to the top of your application to access the NeuroSky.ThinkGear namespace:

```
using NeuroSky.ThinkGear;
```

## Using the NeuroSky.ThinkGear Namespace

The NeuroSky.ThinkGear namespace consists of two classes:

- Connector - Connects to the computer's serial COM port and reads in the port's serial stream of data as DataRowArrays.
- TGParser - Parses a DataRowArray into recognizable ThinkGear [Data Types](#) that your application can use.

To use the classes, first declare a Connector instance and initialize it:

```
private Connector connector;  
connector = new Connector();
```

Next, create EventHandlers to handle each type of [Connector Event](#), and link those handlers to the Connector events.

```
connector.DeviceConnected += new EventHandler( OnDeviceConnected );  
connector.DeviceFound += new EventHandler( OnDeviceFound );  
connector.DeviceNotFound += new EventHandler( OnDeviceNotFound );  
connector.DeviceConnectFail += new EventHandler( OnDeviceNotFound );  
connector.DeviceDisconnected += new EventHandler( OnDeviceDisconnected );  
connector.DeviceValidating += new EventHandler( OnDeviceValidating );
```

In the handler for the DeviceConnected event, you should create another EventHandler to handle DataReceived events from the Device, like this:

```
void OnDeviceConnected( object sender, EventArgs e ) {  
  
    Connector.DeviceEventArgs deviceEventArgs = (Connector.DeviceEventArgs)e;  
    Console.WriteLine( "New Headset Created." + deviceEventArgs.Device.  
DevicePortName );  
  
    deviceEventArgs.Device.DataReceived += new EventHandler( OnDataReceived );  
}
```

Now, whenever data is received from the device, the DataReceived handler will process that data. Here is an example OnDeviceReceived() that shows how it can do this, using a [TGParser](#) to parse the DataRow[]:

```

void OnDataReceived( object sender, EventArgs e ){

    /* Cast the event sender as a Device object, and e as the Device's
    DataEventArgs */
    Device d = (Device)sender;
    Device.DataEventArgs de = (Device.DataEventArgs)e;

    /* Create a TGParser to parse the Device's DataRowArray[] */
    TGParser tgParser = new TGParser();
    tgParser.Read( de.DataRowArray );

    /* Loop through parsed data TGParser for its parsed data... */
    for( int i=0; i<tgParser.ParsedData.Length; i++ ) {

        // See the Data Types documentation for valid keys such
        // as "Raw", "PoorSignal", "Attention", etc.

        if( tgParser.ParsedData[i].ContainsKey("Raw") ){
            Console.WriteLine( "Raw Value:" + tgParser.ParsedData[i]["Raw"] );
        }

        if( tgParser.ParsedData[i].ContainsKey("PoorSignal") ){
            Console.WriteLine( "PQ Value:" + tgParser.ParsedData[i][
"PoorSignal"] );
        }

        if( tgParser.ParsedData[i].ContainsKey("Attention") ) {
            Console.WriteLine( "Att Value:" + tgParser.ParsedData[i][
"Attention"] );
        }

        if( tgParser.ParsedData[i].ContainsKey("Meditation") ) {
            Console.WriteLine( "Med Value:" + tgParser.ParsedData[i][
"Meditation"] );
        }

    }
}

```

When you would like to begin the Mental Effort and/or Familiarity <sup>1)</sup> calculations, use the connector to enable them:

```

connector.setMentalEffortEnable(true);
connector.setTaskFamiliarityEnable(true);

```

Once you have the handlers set up as described above, you can have your Connector actually connect to a device/headset/COM port by using one of the Connect methods described in [Connect to a device](#) below. If the portName is valid and the connection is successful, then your OnDataReceived() method will automatically be called and executed whenever data arrives from the headset.

Before exiting, your application **must** close the Connector's open connections by calling the Connector's `close()` method.

```
connector.close();
```

If `close()` is not called on an open connection, and that connection's process is still alive (i.e. a background thread, or a process that only closed the GUI window without terminating the process itself), then the headset will still be connected to the process, and no other process will be able to connect to the headset until it is disconnected.

## Events

If you choose to connect by stating a specific COM port, it will take the following steps:

1. `connector.Connect(portName);`
2. `connector.Connect` in turn validates the COM port. So the `DeviceValidating` event is triggered
3. if the COM port was valid, it connects to the device. The `DeviceFound` event is never triggered
4. if the COM port was invalid, the `DeviceNotFound` event is triggered.

If you choose to connect by using the AUTO approach, it will take the following steps:

1. `connector.Find();`
2. if it is able to find a COM port with valid ThinkGear Packets, it triggers `DeviceFound`. Otherwise, the `DeviceNotFound` event is triggered
3. the `OnDeviceFound` method in turn calls `connector.Connect(tempPortName);` where `tempPortName` is the valid COM port. This in turn calls `DeviceValidating`.
4. if the COM port was valid, it connects to the device.
5. if the COM port was invalid, the `DeviceNotFound` event is triggered.

## Tips on using ThinkGear.NET

- In order to connect quickly, your application should always remember across sessions the last COM `portName` that was able to successfully connect, and try to connect to that same `portName` first the next time a connection attempt is made. If that remembered `portName` is no longer valid or unable to connect, then you can use `ConnectScan( string portName )` method to find another valid `portName`.
- If an unexpected disconnection occurs, your application should try to reconnect automatically and prompt the user to check their headset device for the following:
  - Battery is properly inserted into the headset device, and has sufficient charge (or try a new battery)
  - Headset device is turned on
  - Headset device is properly paired in Bluetooth settings
  - Headset device is within range of the Bluetooth receiver (within 10m unobstructed)

# API Reference

## Connector class

### Methods

#### Connect to a device

##### **void Connect(string portName)**

Attempts to open a connection with the port name specified by portName. Calling this method results in one of two events being broadcasted:

- DeviceConnected - A connection was successfully opened on portName
- DeviceConnectFail - The connection attempt was unsuccessful

##### **void ConnectScan()**

Attempts to open a connection to the first Device seen by the Connector. Calling this method results in one of two events being broadcasted:

- DeviceConnected - A connection was successfully opened on portName
- DeviceConnectFail - The connection attempt was unsuccessful

##### **void ConnectScan(string portName)**

Same as ConnectScan but scans the port specified by portName first. Calling this method results in one of two events being broadcasted:

- DeviceConnected - A connection was successfully opened on portName
- DeviceConnectFail - The connection attempt was unsuccessful

#### Disconnect from a device

## **void Disconnect()**

Closes all open connections. Calling this method will result in the following event being broadcasted for each open device:

- DeviceDisconnected - The device was disconnected

## **void Disconnect(Connection connection)**

Closes a specific Connection specified by `connection`. Calling this method will result in the following event being broadcasted for a specific open device:

- DeviceDisconnected - The device was disconnected

## **void Disconnect(Device device)**

Closes a specific Device specified by `device`. Calling this method will result in the following event being broadcasted for a specific open device:

- DeviceDisconnected - The device was disconnected

## **Send bytes to a device**

### **void Send(string portName, byte[] bytesToSend)**

Sends an array of bytes to a specific port

## **Configure Mental Effort/Familiarity**

### **void enableMentalEffort() DEPRECATED**

Starts recording data for 60 seconds. Once the recording is complete, the Mental Effort will be calculated. Note: the first time the Mental Effort <sup>2)</sup> is calculated, the result is 0.

### **void enableFamiliarity() DEPRECATED**



Starts recording data for 60 seconds. Once the recording is complete, the Familiarity will be calculated. Note: the first time the Familiarity is calculated, the result is 0.

## **Events**

### **DeviceFound**

Occurs when a ThinkGear device is found. This is where the application chooses to connect to that port or not.

### **DeviceNotFound**

Occurs when a ThinkGear device could not be found. This is usually where the application displays an error that it did not find any device.

### **DeviceValidating**

Occurs right before the connector attempts a serial port. Mainly used to notify the GUI which port it is trying to connect.

### **DeviceConnected**

Occurs when a ThinkGear device is connected. This is where the application links the OnDataReceived for that device.

### **DeviceConnectFail**

Occurs when the Connector fails to connect to that port specified.

### **DeviceDisconnected**

Occurs when the Connector disconnects from a ThinkGear device.

## DataReceived

Occurs when data is available from a ThinkGear device.

## TGParser Class

The TGParser class is used to convert the received data into easily accessible data contained in a Dictionary.

## Methods

### Dictionary<string, double>[] Read( DataRow[] dataRow )

Parses the raw headset data in dataRow and returns a dictionary of usable data. It also stores the dictionary in the ParsedData property.

When connected to a MindWave, or MindWave Mobile headset, the Read ( ) method can return the following standard keys in its dictionary:

| Key            | Description   | Data Type |
|----------------|---|-----------|
| Time           | TimeStamps of packet received   | double    |
| Raw            | Raw EEG data  | short     |
| EegPowerDelta  | Delta Power   | uint      |
| EegPowerTheta  | Theta Power   | uint      |
| EegPowerAlpha1 | Low Alpha Power   | uint      |
| EegPowerAlpha2 | High Alpha Power  | uint      |
| EegPowerBeta1  | Low Beta Power  | uint      |
| EegPowerBeta2  | High Beta Power   | uint      |
| EegPowerGamma1 | Low Gamma Power   | uint      |
| EegPowerGamma2 | High Gamma Power  | uint      |
| Attention      | Attention eSense  | double    |
| Meditation     | Meditation eSense   | double    |
| PoorSignal     | Poor Signal   | double    |
| BlinkStrength  | Strength of detected blink. The Blink Strength ranges from 1 (small blink) to 255 (large blink). Unless a blink occurred, nothing will be returned. Blinks are only calculated if PoorSignal is less than 51. | uint      |

| Key                  | Description  | Data Type |
|----------------------|--|-----------|
| Mental Effort (BETA) | Mental Effort measures how hard the subject's brain is working, i.e. the amount of workload involved in the task. Mental Effort algorithm can be used for both within-trial monitoring (continuous real-time tracking) and between-trial comparison. A trial can be of any length equal to or more than 1 minute. In each trial, the first output index will be given out after the first minute and new output indexes will then be generated at time interval defined by the output rate (default: 10s). | double    |
| Familiarity (BETA)   | Familiarity measures how well the subject is learning a new task or how well his performance is with certain task. Familiarity algorithm can be used for both within-trial monitoring (continuous real-time tracking) and between-trial comparison. A trial can be of any length equal to or more than 1 minute. In each trial, the first output index will be given out after the first minute and new output indexes will then be generated at time interval defined by the output rate (default: 10s).  | double    |

When connected to a ThinkCap, the Read ( ) method can return the following keys in its dictionary:

| Key    | Description                   | Data Type |
|--------|-------------------------------|-----------|
| Time   | TimeStamps of packet received | double    |
| RawCh1 | EEG Channel 1                 | short     |
| RawCh2 | EEG Channel 2                 | short     |
| RawCh3 | EEG Channel 3                 | short     |
| RawCh4 | EEG Channel 4                 | short     |
| RawCh5 | EEG Channel 5                 | short     |
| RawCh6 | EEG Channel 6                 | short     |
| RawCh7 | EEG Channel 7                 | short     |
| RawCh8 | EEG Channel 8                 | short     |

## ThinkGear Data Types

The ThinkGear data types are generally divided into two groups: data types that are only applicable for EEG sensor devices, and data types that are typically applicable to all ThinkGear-based devices.

### General

These data types are generally available from most or all types of ThinkGear hardware devices.

### POOR\_SIGNAL/SENSOR\_STATUS

This integer value provides an indication of how good or how poor the bio-signal is at the sensor. This value is typically output by all ThinkGear hardware devices once per second.

This is an extremely important value for any app using ThinkGear sensor hardware to always read, understand, and handle. Depending on the use cases for your app and users, your app may need to alter the way it uses other data values depending on the current value of `POOR_SIGNAL/SIGNAL_STATUS`. For example, if this value is indicating that the bio-sensor is not currently contacting the subject, then any received `RAW_DATA` or `EEG_POWER` values during that time should be treated as floating noise not from a human subject, and possibly discarded based on the needs of the app. The value should also be used as a basis to prompt the user to possibly adjust their sensors, or to put them on in the first place.

This updated version converts poorSignal values read from different hardware devices. It converts them into a uniform format. (unlike earlier version of the SDK) If you have software that reacts to the poorSignal value, you should evaluate that software to see if changes need to be made

**Poor signal may be caused by a number of different things.** In order of severity, they are:

- Sensor, ground, or reference electrodes not being on a person's head/body
- Poor contact of the sensor, ground, or reference electrodes to a person's skin
- Excessive motion of the wearer (i.e. moving head or body excessively, jostling the headset/sensor).
- Excessive environmental electrostatic noise (some environments have strong electric signals or static electricity buildup in the person wearing the sensor).
- Excessive biometric noise

For EEG modules, a certain amount of noise is unavoidable in normal usage of ThinkGear sensor hardware, and both NeuroSky's filtering technology and algorithms have been designed to detect, correct, compensate for, account for, and tolerate many types of signal noise. Most typical users who are only interested in using the eSense™ values, such as Attention and Meditation, do not need to worry as much about the `POOR_SIGNAL` Quality value, except to note that the Attention and Meditation values will not be updated while `POOR_SIGNAL` is greater than zero, and that the headset is not being worn while `POOR_SIGNAL` is higher than 128. The `POOR_SIGNAL` Quality value is more useful to some applications which need to be more sensitive to noise (such as some medical or research applications), or applications which need to know right away when there is even minor noise detected.

## RAW\_DATA

This data type supplies the raw sample values acquired at the bio-sensor. The sampling rate (and therefore output rate), possible range of values, and interpretations of those values (conversion from raw units to volt) for this data type are dependent on the hardware characteristics of the ThinkGear hardware device performing the sampling. You must refer to the documented development specs of [each type of ThinkGear hardware](#) that your app will support for details.

As an example, the majority of ThinkGear devices sample at 512Hz, with a possible value range of -32768 to 32767.

As another example, to convert TGAT-based EEG sensor values (such as TGAT, TGAM, MindWave, MindWave Mobile) to voltage values, use the following conversion:

```
(rawValue * (1.8/4096)) / 2000
```

## RAW\_MULTI

*This data type is not currently used by any current commercially-available ThinkGear products. It is kept here for backwards compatibility with some end-of-life products, and as a placeholder for possible future products.*

## EEG

These data types are only available from EEG sensor hardware devices, such as the MindWave, MindWave Mobile, MindBand, and TGAM chips and modules.

## ATTENTION

This int value reports the current eSense™ Attention meter of the user, which indicates the intensity of a user's level of mental “focus” or “attention”, such as that which occurs during intense concentration and directed (but stable) mental activity. Its value ranges from 0 to 100. Distractions, wandering thoughts, lack of focus, or anxiety may lower the Attention meter levels. See [eSense Meters](#) below for details about interpreting eSense™ levels in general.

By default, output of this Data Value is enabled. It is typically output once a second.

## MEDITATION

This unsigned one-byte value reports the current eSense™ Meditation meter of the user, which indicates the level of a user's mental “calmness” or “relaxation”. Its value ranges from 0 to 100. Note that Meditation is a measure of a person's **mental** levels, not **physical** levels, so simply relaxing all the muscles of the body may not immediately result in a heightened Meditation level. However, for most people in most normal circumstances, relaxing the body often helps the mind to relax as well. Meditation is related to reduced activity by the active mental processes in the brain, and it has long been an observed effect that closing one's eyes turns off the mental activities which process images from the eyes, so closing the eyes is often an effective method for increasing the Meditation meter level. Distractions, wandering thoughts, anxiety, agitation, and sensory stimuli may lower the Meditation meter levels. See [eSense Meters](#) below for details about interpreting eSense™ levels in general.

By default, output of this Data Value is enabled. It is typically output once a second.

## eSense Meters

For all the different types of eSense™ (i.e. Attention, Meditation), the meter value is reported on a relative eSense™ scale of 1 to 100. On this scale, a value between 40 to 60 at any given moment in time is considered “neutral”, and is similar in notion to “baselines” that are established in conventional EEG measurement techniques (though the method for determining a ThinkGear baseline is proprietary and may differ from conventional EEG). A value from 60 to 80 is considered “slightly elevated”, and may be interpreted as levels being possibly higher than normal (levels of Attention or Meditation that may be higher than normal for a given person). Values from 80 to 100 are considered “elevated”, meaning they are strongly indicative of heightened levels of that eSense™.

Similarly, on the other end of the scale, a value between 20 to 40 indicates “reduced” levels of the eSense™, while a value between 1 to 20 indicates “strongly lowered” levels of the eSense™. These levels may indicate states of distraction, agitation, or abnormality, according to the opposite of each eSense™.

## BLINK

This int value reports the intensity of the user's most recent eye blink. Its value ranges from 1 to 255 and it is reported whenever an eye blink is detected. The value indicates the relative intensity of the blink, and has no units.

The Detection of Blinks must be enabled.

```
if (setBlinkDetectionEnabled(true)) {  
    // return true, means success  
  
    Console.WriteLine("HelloEEG: BlinkDetection is Enabled");  
}  
else {  
    // return false, meaning not supported because:  
    // + connected hardware doesn't support  
    // + conflict with another option already set  
    // + not support by this version of the SDK  
  
    Console.WriteLine("HelloEEG: BlinkDetection can not be Enabled");  
}
```

The current configuration can be retrieved.

```
if (getBlinkDetectionEnabled()) {  
    // return true, means it is enabled  
  
    Console.WriteLine("HelloEEG: BlinkDetection is configured");  
}
```

```
else {  
    // return false, meaning not currently configured  
  
    Console.WriteLine("HelloEEG: BlinkDetection is NOT configured");  
}
```

If these methods are called before the MSG\_MODEL\_IDENTIFIED has been received, it is considered a request to be processed when the connected equipment is identified. It is possible to Enable this feature and later find that it is no longer enabled. Once the connected equipment has been identified, if the request is incompatible with the hardware or software it will be overridden and the MSG\_ERR\_CFG\_OVERRIDE message sent to provide notification.

## EEG\_POWER

This Data Value represents the current magnitude of 8 commonly-recognized types of EEG frequency bands.

The eight EEG powers are: delta (0.5 - 2.75Hz), theta (3.5 - 6.75Hz), low-alpha (7.5 - 9.25Hz), high-alpha (10 - 11.75Hz), low-beta (13 - 16.75Hz), high-beta (18 - 29.75Hz), low-gamma (31 - 39.75Hz), and mid-gamma (41 - 49.75Hz). These values have no units and are only meaningful for comparison to the values for the other frequency bands within a sample.

By default, output of this Data Value is enabled, and it is output approximately once a second.

## THINKCAP\_RAW

*This data type is not currently used by any current commercially-available ThinkGear products. It is kept here for backwards compatibility with some end-of-life products, and as a placeholder for possible future products.*

## MENTAL EFFORT (BETA)

Mental Effort is included in a Beta form. The Mental Effort Algorithm is not a formal inclusion of the Developer Tools 2.5 release but can still be tested in your application design. Questions or feedback? Email: [support@neurosky.com](mailto:support@neurosky.com)

For the most up to date versions of this algorithm and the corresponding documentation, please refer to <http://developer.neurosky.com> and any specific new algorithm preview packages.

When applied to single-channel EEG data collected from forehead area, the Mental Effort algorithm measures the amount of workload exerted by subject's brain while performing a task (how hard the

subject's brain is working while performing the task). The Mental Effort algorithm works well with both motor (e.g. drawing) and mental (e.g. reciting) tasks. The algorithm can be used for monitoring a subject's Mental Effort curve and changes in realtime (to see how their Mental Effort Index changes as they're performing the task), or it can be used for studying their Mental Effort levels on a task across separate sessions or days.

The Mental Effort algorithm must be applied to at least 1 minute of EEG data. A Mental Effort Index will be output by the algorithm after the first 60s, and then every N seconds after that (N is the predefined output rate; default: 10s).

A typical way to use the Mental Effort algorithm is for an application to prompt a subject to be relaxing, with their eyes open, doing nothing, for the first 60s while taking the initial measurement. The first Mental Effort Index value (reported after the first 60s) is kept by the application as a baseline, against which subsequent Mental Effort Index values (by default received every 10s) can be compared to determine relative percent changes (e.g. after relaxing for the first minute, the user starts doing a specific task. Their Mental Effort Index decreases by -20% in the first 10s of the task compared to the first minute baseline, and then goes down further to -35% in the next 10s of the task compared to the first minute baseline, indicating their brain is doing less and less work in the first 20s. After 40s, their Mental Effort Index goes back up to -5% from the first minute baseline, which may indicate their brain is starting to do more work again at that point.)

Alternatively, another possible baseline measurement could be, instead of relaxing and doing nothing during the first 60s, to rather have the subject start doing the task right away and regard the initial part of the task as the baseline. You would then interpret the percentage changes accordingly, knowing that the baseline is based on the initial 60s engaging in the task.

For presenting, studying, and interpreting the data, each reported Mental Effort Index (MEI) value can be compared for percent changes either (1) against the baseline value, or (2) against the MEI value immediately preceding it.

The Mental Effort Index is a floating point number with arbitrary units. This means the MEI numbers have no meaning on their own, and take on meaning only when comparing percentage changes between two or more values.

An example on how to use the algorithm could be found in the sample application - "HelloEEG".

A few example tasks that the algorithm can to: arithmetic calculation ("Math 24" available in <http://www.24theory.com/>, and "Addition Aliens Attack" available in <http://www.imathgame.com/ChromeAddition.php>>)

To use the Mental Effort algorithm <sup>3)</sup> in the NeuroSky SDK/API library, it must first be enabled:

```
if (setMentalEffortEnable(true)) {  
    // return true, means success  
  
    Console.WriteLine("HelloEEG: MentalEffort is Enabled");  
}  
else {  
    // return false, meaning not supported because:  
    // + connected hardware doesn't support  
    // + conflict with another option already set
```



```
// + not support by this version of the SDK

Console.WriteLine("HelloEEG: MentalEffort can not be Enabled");
}
```

At any time, the status of the algorithm (whether it is enabled) can be queried:

```
if (getMentalEffortEnable()) {
    // return true, means it is enabled

    Console.WriteLine("HelloEEG: MentalEffort is configured");
}
else {
    // return false, meaning not currently configured

    Console.WriteLine("HelloEEG: MentalEffort is NOT configured");
}
```

Using just `setMentalEffortEnable(true)`, the algorithm will be executed exactly one time. The execution begins as soon as 60 seconds of good data has been collected. After the results have been reported, the algorithm becomes automatically disabled.

It is possible to configure the algorithm to run continuously:

```
if (setMentalEffortRunContinuous(true)) {
    // return true, means success

    Console.WriteLine("HelloEEG: MentalEffort Continuous operation");
}
else {
    // return false, meaning not supported because:
    // + connected hardware doesn't support
    // + conflict with another option already set
    // + not support by this version of the SDK

    Console.WriteLine("HelloEEG: MentalEffort normal operation ");
}
```

Enabling Continuous Mode does not automatically also enable the algorithm itself; you must still call `setMentalEffortEnable(true)` yourself.

The current Continuous Mode configuration can be retrieved.

```
if (getMentalEffortRunContinuous()) {
    // return true, means it is enabled

    Console.WriteLine("HelloEEG: MentalEffort Continuous operation");
}
else {
```

```
// return false, meaning not currently configured
```

```
Console.WriteLine("HelloEEG: MentalEffort normal operation");  
}
```

If these methods are called before the MSG\_MODEL\_IDENTIFIED has been received, it is considered a request to be processed when the connected equipment is identified. It is possible to Enable this feature and later find that it is no longer enabled. Once the connected equipment has been identified, if the request is incompatible with the hardware or software it will be overridden and the MSG\_ERR\_CFG\_OVERRIDE message sent to provide notification.

This algorithm is resource and computation intensive. If you need to run with the Debugger, be aware that this calculation may take many minutes to complete when the debugger is engaged. It will output its results only after its calculations are complete.

## FAMILIARITY (BETA)

Familiarity is included in a Beta form. The Familiarity Algorithm is not a formal inclusion of the Developer Tools 2.5 release but can still be tested in your application design. Questions or feedback? Email: [support@neurosky.com](mailto:support@neurosky.com)

When applied to single-channel EEG data collected from the forehead area, the Familiarity algorithm measures how well a subject is learning a task, and correlates with how well his performance is with certain tasks. The Familiarity algorithm works well with both motor (e.g. drawing) and mental (e.g. reciting) tasks. The algorithm can be used for monitoring a subject's mental Familiarity curve and changes in realtime (to see how their Familiarity Index changes as they're performing the task), or it can be used for studying their Familiarity levels on a task across separate sessions or days.

The Familiarity algorithm must be applied to at least 1 minute of EEG data. A Familiarity Index will be output by the algorithm after the first 60s, and then every N seconds after that (N is the predefined output rate; default: 10s).

A typical way to use the Familiarity algorithm is for an application to prompt a subject to be relaxing, with their eyes open, doing nothing, for the first 60s while taking the initial measurement. The first Familiarity Index value (reported after the first 60s) is kept by the application as a baseline, against which subsequent Familiarity Index values (by default received every 10s) can be compared to determine relative percent changes (e.g. after relaxing for the first minute, the user starts doing a specific task. Their Familiarity Index goes up to +20% in the first 10s of the task compared to the first minute baseline, and then goes up to +35% in the next 10s of the task compared to the first minute baseline. After 40s, their Familiarity Index may dip to -15% from the first minute baseline, which may indicate they are no longer learning and absorbing the task as well, or may no longer be performing optimally at the task.)

Alternatively, another possible baseline measurement could be, instead of relaxing and doing nothing during the first 60s, to rather have the subject start doing the task right away and regard the initial part of the task as the baseline. You would then interpret the percentage changes accordingly, knowing that the baseline is based on the initial 60s engaging in the task.

For presenting, studying, and interpreting the data, each reported Familiarity Index (FI) value can be compared for percent changes either (1) against the baseline value, or (2) against the FI value immediately preceding it.

The Familiarity Index is a floating point number with arbitrary units. This means the FI numbers have no meaning on their own, and take on meaning only when comparing percentage changes between two or more values.

A few example tasks that the algorithm can to: painting ("Color Pages" available in < <http://coloringkid.net/>>) and dancing ("Just Dance" available in < <http://www.youtube.com/watch?v=4Hh7FY3n3wM>>).

To use the Familiarity algorithm in the NeuroSky SDK/API library, it must first be enabled:

```
if (setTaskFamiliarityEnable(true)) {  
    // return true, means success  
  
    Console.WriteLine("HelloEEG: Familiarity is Enabled");  
}  
else {  
    // return false, meaning not supported because:  
    // + connected hardware doesn't support  
    // + conflict with another option already set  
    // + not support by this version of the SDK  
  
    Console.WriteLine("HelloEEG: Familiarity can not be Enabled");  
}
```

At any time, the status of the algorithm (whether it is enabled) can be queried:

```
if (getTaskFamiliarityEnable()) {  
    // return true, means it is enabled  
  
    Console.WriteLine("HelloEEG: TaskFamiliarity is configured");  
}  
else {  
    // return false, meaning not currently configured  
  
    Console.WriteLine("HelloEEG: TaskFamiliarity is NOT configured");  
}
```

Using just `setTaskFamiliarityEnable(true)`, the algorithm will be executed exactly one time. The execution begins as soon as 60 seconds of good data has been collected. After the results have been reported, the algorithm becomes automatically disabled.

It is possible to configure the algorithm to run continuously:

```
if (setTaskFamiliarityRunContinuous(true)) {  
    // return true, means success  
  
    Console.WriteLine("HelloEEG: Familiarity Continuous operation");  
}  
else {  
    // return false, meaning not supported because:  
    // + connected hardware doesn't support  
    // + conflict with another option already set  
    // + not support by this version of the SDK  
  
    Console.WriteLine("HelloEEG: Familiarity normal operation ");  
}
```

Enabling Continuous Mode does not automatically also enable the algorithm itself; you must still call `setTaskFamiliarityEnable(true)` yourself.

The current Continuous Mode configuration can be retrieved:

```
if (getTaskFamiliarityRunContinuous()) {  
    // return true, means it is enabled  
  
    Console.WriteLine("HelloEEG: Familiarity Continuous operation");  
}  
else {  
    // return false, meaning not currently configured  
  
    Console.WriteLine("HelloEEG: Familiarity normal operation");  
}
```

If these methods are called before the `MSG_MODEL_IDENTIFIED` has been received, it is considered a request to be processed when the connected equipment is identified. It is possible to Enable this feature and later find that it is no longer enabled. Once the connected equipment has been identified, if the request is incompatible with the hardware or software it will be overridden and the `MSG_ERR_CFG_OVERRIDE` message sent to provide notification.

This algorithm is resource and computation intensive. If you need to run with the Debugger, be aware that this calculation may take many minutes to complete when the debugger is engaged. It will output its results only after its calculations are complete.

# Proper App Design

Before releasing an app for real-world use, make sure your app considers or handles the following:

- If your app's Handler receives a `MSG_STATE_CHANGE` Message with any value other than `STATE_CONNECTING` or `STATE_CONNECTED`, it should carefully handle each possible error situation with an appropriate message to the user via the app's UI. Not handling these error cases well in the UI almost always results in an extremely poor user experience of the app. Here are some examples:
  - If a `STATE_ERR_BT_OFF` Message is received, the user should be prompted to turn on their Bluetooth adapter, and then they can try again.
  - If a `STATE_ERR_NO_DEVICE` Message is received, the user should be reminded to first pair their ThinkGear hardware device to their Android device's Bluetooth, according to the instructions they received with their ThinkGear hardware device.
  - If a `STATE_NOT_FOUND` Message is received, the user should be reminded to check that their ThinkGear hardware device is properly paired to their Android device (same as the `STATE_ERR_NO_DEVICE` case), and if so, that their ThinkGear hardware device is turned on, in range, and has enough battery or charge.
  - See [TGDevice States](#) for more info.
- Always make sure your app is handling the [POOR SIGNAL/SENSOR STATUS](#) Data Type. It is output by almost all ThinkGear devices, and provides important information about whether the sensor is properly in contact with the user. If it is indicating some sort of problem (`problem == not 0`), then your app should notify the user to properly wear the ThinkGear hardware device, and/or disregard any other reported data values while the [POOR SIGNAL/SENSOR STATUS](#) continues to indicate a problem, as appropriate for your app.
- To make the user experience consistent, familiar, and easy-to-learn and use for end customers across platforms and devices, your app should be designed to follow the guidelines and conventions described in NeuroSky's [App Standards](#).

## Troubleshooting

There are currently no known issues. If you encounter any bugs or issues, please visit <http://support.neurosky.com>, or contact [support@neurosky.com](mailto:support@neurosky.com).

If you need further help, you may visit <http://developer.neurosky.com> to see if there is any new information.

To contact NeuroSky for support, please visit <http://support.neurosky.com>, or send email to [support@neurosky.com](mailto:support@neurosky.com).

For developer community support, please visit our community forum on <http://www.linkedin.com/groups/NeuroSky-Brain-Computer-Interface-Technology-3572341>

## Important Notices

The algorithms included in this SDK are solely for promoting the awareness of personal wellness and health and are not a substitute for medical care. The algorithms are not to be used to diagnose, treat, cure or prevent any disease, to prescribe any medication, or to be a substitute for a medical device or treatment. In some circumstances, the algorithm may report false or inaccurate results. The descriptions of the algorithms or data displayed in the SDK documentation, are only examples of the particular uses of the algorithms, and NeuroSky disclaims responsibility for the final use and display of the algorithms internally and as made publically available.

The algorithms may not function well or may display accurate data if the user has a pacemaker.

## 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.

- <sup>1)</sup> you may see older documents refer to this as "Task Difficulty", which is the former name of the Mental Effort algorithm
- <sup>2)</sup> older documents refer to "Task Difficulty", this name is replaced by "Mental Effort" which more plainly describes the functionality
- <sup>3)</sup> older documents may refer to the "Mental Effort" algorithm as "Task Difficulty". The algorithm was renamed because "Mental Effort" more accurately described what it was measuring and calculating.

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

Permanent link:  
[http://developer.neurosky.com/docs/doku.php?id=thinkgear.net\\_sdk\\_dev\\_guide\\_and\\_api\\_reference](http://developer.neurosky.com/docs/doku.php?id=thinkgear.net_sdk_dev_guide_and_api_reference)

Last update: **2014/07/01 20:49**



**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.