ThinkGear SDK for .NET: Development Guide and API Reference

May 30, 2013

NeuroSky
Brain-Computer Interface Technologies

The NeuroSky® product families consist of hardware and software components for simple integration of this biosensor technology into consumer and industrial end-applications. All products are designed and manufactured to meet consumer thresholds for quality, pricing, and feature sets. NeuroSky sets itself apart by providing building block component solutions that offer friendly synergies with related and complementary technological solutions.

Contents

Introduction

This guide will teach you how to use NeuroSky's **ThinkGear SDK for .NET** to write Windows apps that can utilize bio-signal data from NeuroSky's ThinkGear family of bio-sensors (which includes the CardioChip family of products). This will enable your Windows apps to receive and use bio-signal data such as EEG and ECG/EKG acquired from NeuroSky's sensor hardware.

This guide (and the entire **ThinkGear SDK for .NET** for that matter) 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 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 NeuroSky's **ThinkGear SDK for .NET**. Chances are, if you're reading this document, then you already have it.

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
- **TG-HelloEEG.exe** - a reference build of the HelloEEG sample project
- **HelloEEG Sample Project** source code

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

You'll find the source code of the "HelloEEG Sample Project" in the `Sample Projects/HelloEEG` folder.

Supported ThinkGear Hardware

The ThinkGear SDK for .NET must be used with a ThinkGear-compatible hardware sensor device. The following ThinkGear-compatible hardware devices are currently supported:

- MindWave Mobile
- MindWave (RF)

- MindBand
- MindSet
- ThinkCap
- CardioChip Starter Kit Unit
- TGAM module
- CardioChip BMD101 module
- TGAT ASIC
- BMD101 ASIC

Important: Before using any Windows app that uses the TG-SDK for .NET, make sure you have paired the ThinkGear sensor hardware to your Windows machine by carefully following the instructions in the User Manual that came with each ThinkGear hardware device! The ThinkGear sensor must 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 ThinkGear device. Add the project to your Visual Studio by following these steps:

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"
16. select it and right click, and remove the reference to "Microsoft.CSharp"
17. select the "References" section, right click, pick "Add Reference.."
18. choose the browse TAB, choose the folder "neurosky" and then pick "ThinkGear.dll"
19. at the Toolbar select **Build** —> **Build Solution**
20. 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.

Note: These steps have been tested with Visual Studio 2010, if yours is different you may have to adapt these instructions.

Note: 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 /neuosky folder into the .exe so that it can function in a more standalone way.

Developing Your Own ThinkGear-enabled Apps for .NET

Preparing Your .NET Project

The **ThinkGear .NET SDK**'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 .NET SDK**'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 `EventHandler`s 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"] );
        }

        if( tgParser.ParsedData[i].ContainsKey("RespiratoryRate") ){
            Console.WriteLine( "Respiratory Rate:" + tgParser.ParsedData[i]["RespiratoryRate"]);
        }
        if( tgParser.ParsedData[i].ContainsKey("RelaxationLevel") ){
            Console.WriteLine( "Relaxation Level:" + tgParser.ParsedData[i]["RelaxationLevel"]);
        }
    }
}
```

When you would like to begin the Task Familiarity and/or Mental Effort [1] calculations, use the `connector` to enable them:

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

[1] older documents refer to "Task Difficulty", this name is replaced by "Mental Effort" which more plainly describes the functionality

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 Task Familiarity/Mental Effort

void enableTaskDifficulty() DEPRICATED Starts recording data for 60 seconds. Once the recording is complete, the Task Difficulty will be calculated. Note: the first time the Mental Effort [1] is calculated, the result is 0.

void enableTaskFamiliarity() DEPRICATED Starts recording data for 60 seconds. Once the recording is complete, the Task Familiarity will be calculated. Note: the first time the Task 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.

[1] older documents refer to "Task Difficulty", this name is replaced by "Mental Effort" which more plainly describes the functionality

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 MindSet, 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
Zone	performance Zone	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
Task Familiarity	Can be used to compare a test subjects familiarity with a newly learned (motor) skill. One minute of collected data could constitute a trial, and will produce a familiarity index value. The familiarity index of separate trials of the skill can be compared.	double
Mental Effort	Can be used to compare the mental effort needed by a test subject. One minute of collected data could constitute a trial, and will produce a mental effort index value. The mental effort index of separate trials can be compared.	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

When connected to a BMD10X device, the `Read()` method can return the following keys in its dictionary:

Key	Description	Data Type
PoorSignal	Poor Signal/Signal Status	short
Raw	Minimally processed sample from AD converter	short
HeartRate	User's instantaneous heart rate (BPM)	double
RrInt	Time between detected heart beats in milliseconds	uint
RespiratoryRate	User's respiration rate in breaths per minutes	double
Relaxation	User's relaxation level derived from the user's EKG	uint

Detailed Descriptions

ZONE

This value reports the current performance Zone of the subject. It's value ranges from 0 to 9. And this value is sent only when the subject transitions from one Zone to another.

This algorithm uses the Attention and Mediation values to guide a subject to their best performance.

To be in the Elite Zone (9), the subject must hold their Attention level a value of at least 94 and simultaneously holding their Meditation level steady or increasing.

To be in the Intermediate Zone (5), the subject must hold their Attention level a value of at least 64 and simultaneously holding their Meditation level steady or increasing.

To be in the Beginner Zone (1), the subject must hold their Attention level a value of at least 28 and simultaneously holding their Meditation level steady or increasing.

The Not Ready Zone (0) is all Attention levels below 28 and subjects from the Beginner Zone whose Meditation levels are decreasing.

All Zone calculations aresuspended and values reset if the sensor doesn't appear to be in good contact with a human.

Note: This is a different implementation of performance Zone compared to other NeuroSky products. Reference: Golf Putting Training Algorithm v 2.0, September 2012, Dr. KooHyoung Lee.

ThinkGear Data Types

The ThinkGear data types are generally divided into three groups: data types that are only applicable for EEG sensor devices, types that are only applicable for ECG/EKG (CardioChip) sensor devices, and data types that are typically applicable to all ThinkGear-based devices, including EEG and ECG/EKG.

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.

Important: 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 (i.e. unwanted EMG, EKG/ECG, EOG, EEG, etc. signals)

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 Mobile, MindWave, MindSet) to voltage values, use the following conversion:

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

Note that ECG/EKG raw values from CardioChip/BMD10X-based devices must use a different conversion.

```
(rawValue * 18.3) / 128.0
```

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 Mobile, MindSet, 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™.

ZONE

This value reports the current performance Zone of the subject. It's value ranges from 0 to 9. And this value is sent only when the subject transitions from one Zone to another.

This algorithm uses the Attention and Mediation values to guide a subject to their best performance.

To be in the Elite Zone (9), the subject must hold their Attention level a value of at least 94 and simultaneously holding their Meditation level steady or increasing.

To be in the Intermediate Zone (5), the subject must hold their Attention level a value of at least 64 and simultaneously holding their Meditation level steady or increasing.

To be in the Beginner Zone (1), the subject must hold their Attention level a value of at least 28 and simultaneously holding their Meditation level steady or increasing.

The Not Ready Zone (0) is all Attention levels below 28 and subjects from the Beginner Zone whose Meditation levels are decreasing.

All Zone calculations are suspended and values reset if the sensor doesn't appear to be in good contact with a human.

Note: This is a different implementation of performance Zone compared to other NeuroSky products. Reference: Golf Putting Training Algorithm v 2.0, September 2012, Dr. KooHyoung Lee.

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");
}
```

Note: 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.

POSITIVITY

Values -100 to +100, indicates that the subject is attentive, the more negative values mean the subject is less attentive and the more positive values mean more attentive.

Note: This feature is not currently available.

FAMILIARITY

This algorithm seeks to represent the subject's Familiarity with a motor skill, it may be used together with the Mental Effort algorithm to examine different aspects of learned motor skills. But it may also be used independently.

It can be used to compare a test subjects familiarity with a newly learned motor skill. One minute of collected data constitutes a trial, and will produce a familiarity index value for this individual. The familiarity index of separate trials of the motor skill can be compared for the same individual.

The familiarity index is a reported as a floating point number. They have no units and are only meaningful in comparison to other values collected from the same individual. A useful presentation is to calculate the percentage change from a baseline (or the last) trial and the current trial. Note that the difference may be positive or negative. Examine the HelloEEG sample application for one example of how to use this information.

The Calculation of Task Familiarity must be enabled.

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

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

The current configuration can be retrieved.

```
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");
}
```

After it is enabled, this algorithm will be executed one time. The execution begins as soon as 60 seconds of good data has been collected. After the results have been reported, the algorithm is automatically disabled.

It is possible to configure the algorithm to run continuously. But enabling continuous operation does not automatically enable the algorithm, after setting RunContinuous, you must also enable the algorithm.

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

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

The current configuration can be retrieved.

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

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

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

Note: 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.

Note: 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 complete and present it's results. Without the debugger engaged, this calculation should complete in a few seconds.

MENTAL EFFORT

This algorithm seeks to represent the subject's Mental Effort, it may be used together with the Familiarity algorithm to examine different aspects of learned motor skills. But it may also be used independently.

It can be used to compare how difficult a test subjects finds a newly learned motor skill. One minute of collected data constitutes a trial, and will produce a mental effort index value for this individual. The index of separate trials of the motor skill can be compared for the same individual.

The mental effort index is a reported as a floating point number. They have no units and are only meaningful in comparison to other values collected from the same individual. A useful presentation is to calculate the percentage change from an initial (or baseline) trial and the current trial. Note that the difference may be positive or negative. Examine the HelloEEG sample application for one example of how to use this information.

The Calculation of Mental Effort [1] must 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");
}
```

The current configuration can be retrieved.

```
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");
}
```

After it is enabled, this algorithm will be executed one time. The execution begins as soon as 60 seconds of good data has been collected. After the results have been reported, the algorithm is automatically disabled.

It is possible to configure the algorithm to run continuously. But enabling continuous operation does not automatically enable the algorithm, after setting RunContinuous, you must also enable the algorithm.

```
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 ");
}
```

[1] older documents refer to "Task Difficulty", this name is replaced by "Mental Effort" which more plainly describes the functionality

The current 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");
}
```

Note: 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.

Note: 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 complete and present it's results. Without the debugger engaged, this calculation should complete in a few seconds.

ECG/EKG

These data types are only available from ECG/EKG sensor (CardioChip) hardware devices, such as the CardioChip Starter Kit Unit and BMD10X chips and modules.

HeartRate

This int value reports the current heart rate of the user, in units of beats per minute (BPM). Unlike many other commonly seen reports of heart rate from other devices, this value is calculated precisely in real time based on the actual time between each and every one of the user's actual R-peaks. This results in a very precise and continuous reporting of Heart Rate that changes with the actual beat-to-beat fluctuations of every single one of the user's actual heart beats.

To easily get an "smoothed, averaged" heart rate value which is more commonly seen as reported by other ECG/EKG devices, use these values as inputs to the Smoothed Heart Rate described below.

Smoothed Heart Rate

Typically, when viewing a "Heart Rate" value on many ECG/EKG devices, a "smoothed" value is displayed so that there aren't rhythmic fluctuations in the viewed heart rate based on the subject's natural HRV rhythms. The same sort of "smoothed" effect can be achieved against the precise HEART RATE values, by using the `getAcceleration()` method of the `HeartRateAcceleration` class provided in this SDK.

See the section on Heart Rate Acceleration for a description of how to calculate the Smoothed Heart Rate, and then refer to the API Reference for full details on the `HeartRateAcceleration` class.

Heart Rate Acceleration

A potentially useful metric of Heart Rate is the acceleration rate. A positive acceleration value indicates the user's heart rate is speeding up by a certain number of BPM over a given period of time (such as over 10 seconds), while a negative acceleration value indicates the user's heart rate is slowing down by a certain number of BPM over a given period. When starting exercise, or during rest after exercise, this acceleration metric could be used as an indicator of how quickly a person's heart is speeding up to match the activity, or how quickly it is slowing down back to normal, respectively.

To calculate Heart Rate Acceleration (and/or Smoothed Heart Rate), first initialize a HeartRateAcceleration() object in your app:

```
HeartRateAcceleration heartRateAcceleration = new HeartRateAcceleration();
```

This will initialize the calculation to use a period of 10 second. (You can instead choose to use the overloaded constructors to initialize the calculation using a longer or shorter period of time, as appropriate for your app).

Then, whenever a new Heart Rate value becomes available to your app, get the Smoothed Heart Rate and acceleration values like this:

```
int[] result = heartRateAcceleration.getAcceleration( heartRate, poorSignal );
if( result[0] != -1 ) {
    int smoothedHeartRate = result[0];
    int heartRateAcceleration = result[1];
}
```

Refer to the API Reference documentation for full details of the `HeartRateAcceleration` class.

Target Heart Rate for Physical Training

Given information about a user's age and gender, it is possible to determine a target range of heart rates for them to achieve particular physical training "zones". Combined with the HEART RATE information reported by the sensor, an app could advise a user whether their current heart rate is within their target training zone (such as right after a workout).

To determine the target range of heart rates for a person, first create a `TargetHeartRate` object:

```
TargetHeartRate targetHeartRate = new TargetHeartRate();
```

Then, at any time, to determine the target range of heart rates (min to max values) for a a user to achieve a particular physical training zone, use the `getTargetHeartRate()` method:

```
int age = 25;
String gender = "Male";
String zone = "Aerobic";
int[] range = targetHeartRate.getTargetHeartRate( age, gender, zone );

int lowerBound = range[0];
int upperBound = range[1];
```

The `lowerBound` and `upperBound` could then be compared to the user's HEART RATE or Smoothed Heart Rate to determine if the user is within the their target range for the target physical training zone.

The `gender` must be either `"Male"` or `"Female"`. The `zone` must be one of:

- `"Light Exercise"`
- `"Weight Loss"`
- `"Aerobic"`
- `"Conditioning"`
- `"Athletic"`

If any of the arguments are incorrect, then the method will return an int[] of `-1, -1` .

Important: The heart rate should not be measured while the user is engaged in physical activity; the user should temporarily stop the activity and then measure their heart rate.

(References)

1. http://www.heart.org/HEARTORG/GettingHealthy/PhysicalActivity/Target-Heart-Rates_UCM_434341_
2. http://www.cdc.gov/physicalactivity/everyone/measuring/heartrate.html
3. http://www.heart.com/heart-rate-chart.html
4. http://www.thewalkingsite.com/thr.html

Heart Fitness Level

Given a person's age, gender, and current resting heart rate, it is possible to get a general idea of the person's current heart health and fitness, labeling them as one of "Poor", "Below Average", "Average", "Above Average", "Good", "Excellent", or "Athlete".

To determine the heart fitness level for a person, first create a `HeartFitnessLevel` object:

```
HeartFitnessLevel heartFitnessLevel = new HeartFitnessLevel();
```

Then, once you have the age, gender, and current *resting* heart rate of the person, use the `getHeart-FitnessLevel()` method:

```
int age = 25;
String gender = "Male";  // "Male" or "Female"
int restingHR = 60;
String heartFitnessLevel = heartFitnessLevel.getHeartFitnessLevel( age, gender, restingHR );
```

The gender must be one of "Male" or "Female", otherwise the method will simply return the empty string ("").

The `heartFitnessLevel` will be returned as one of "Poor", "Below Average", "Average", "Above Average", "Good", "Excellent", or "Athlete".

(References)

1. http://www.topendsports.com/testing/heart-rate-resting-chart.htm

RELAXATION

The Relaxation data value gives an indication of whether a user's heart is showing indications of relaxation, or is instead showing indications of excitation, stress, or fatigue, based on the user's Heart Rate Variability (HRV) characteristics. It is reported on a scale from 1 to 100. High Relaxation values tend to indicate a state of relaxation, while low values tend to indicate excitation, stress, or fatigue.

To receive these values via MSG_RELAXATION messages to your app's Handler, simply have the TGDevice connected to a ThinkGear ECG/EKG sensor (CardioChip), and a user contacting the ECG/EKG sensor hardware properly for at least one minute continuously with a good, clean signal (SENSOR_STATUS == 200 for 1 minute). If the signal is interrupted, and SENSOR_STATUS becomes anything other than 200, then this calculation is reset and starts over, requiring another minute of clean data to report a MSG_RELAXATION.

For best results, the user should be sitting calmly during data collection.

(References)

1. Neurosci Biobehav Rev. 2009 Feb; 33(2): 71-80. Epub 2008 Jul 30. Heart rate variability explored in the frequency domain: a tool to investigate the link between heart and behavior. Montano N, Porta A, Cogliati C, Costantino G, Tobaldini E, Casali KR, Iellamo F.
2. Int J Cardiol. 2002 Jul; 84(1): 1-14. Functional assessment of heart rate variability: physiological basis and practical applications. Pumprla J, Howorka K, Groves D, Chester M, Nolan J.
3. International Conference on Computer and Automation Engineering. A Review of Measurement and Analysis of Heart Rate Variability. Dipali Bansal, Munna Khan, A. K. Salhan.
4. Neurosci Biobehav Rev. 2009 Feb; 33(2): 81-8. Epub 2008 Aug 13. Claude Bernard and the heart-brain connection: further elaboration of a model of neurovisceral integration. Thayer JF, Lane RD.

Respiratory Rate

The Respiration data value reports a user's approximate respiration rate in breaths per minute. It is calculated from the user's ECG/EKG and Heart Rate Variability (HRV) characteristics.

The respiration rate is received by the application through the `RespiratoryRate` key, as in this example code:

```
if( tgParser.ParsedData[i].ContainsKey("RespiratoryRate") ){
    Console.WriteLine( "Respiratory Rate:" + tgParser.ParsedData[i]["RespiratoryRate"]);
}
```

For best results, the user should be sitting calmly during data collection.

(References)

1. Rosenthal, Talma, Ariela Alter, Edna Peleg, and Benjamin Gavish. "Device-guided breathing exercises reduce blood pressure: ambulatory and home measurements." American Journal of Hypertension. 14. (2001): 74–76.

The Calculation of Respiration Rate must be enabled. And once enabled it will run continuously until disabled.

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

     Console.WriteLine( "HelloEKG: RespirationRate 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( "HelloEKG: RespirationRate can not be Enabled");
}
```

The current configuration can be retrieved.

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

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

     Console.WriteLine( "HelloEKG: RespirationRate is NOT configured");
}
```

Note: 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.

Note: 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 complete and present it's results. Without the debugger engaged, this calculation should complete in a few seconds.

Heart Risk Awareness

The Heart Risk Awareness data value aims to raise awareness if the HRV is very low, as low HRV has been shown to be associated with increased risk of mortality.

To determine the Heart Risk Awareness for a person, first create a NeuroSkyHeartMeters object:

```
    NeuroSkyHeartMeters neuroSkyHeartMeters = new NeuroSkyHeartMeters();
```

Then, use one of the following two methods to calculate:

Using R-R Intervals Collection

Whenever your app's Handler receives a MSG_EKG_RRINT Message, save the R-R Interval value into a buffer. Once you have at least 60 R-R Intervals in the buffer, use the `calculateHeartRiskAware( Integer[] rrIntervalInMS )` method of the `NeuroSkyHeartMeters` class:

```
    private final Handler handler = new Handler() {

        ArrayList<Integer> temp_rrintBuffer = new ArrayList<Integer>();
        Integer[] rrinterBuffer = new Integer[60];

        @Override
        public void handleMessage( Message msg ) {

            switch( msg.what ) {

              //...

              case MSG_EKG_RRINT:
                  temp_rrintBuffer.add( msg.arg1 );
                  if( temp_rrintBuffer.size()==60 ) {
                      for( int i = 0; i<60; i++ ) {
                          rrintBuffer[i] = temp_rrintBuffer.get(i);
                      }
                      temp_rrintBuffer.clear();
                      int heartRiskAwareness = neuroSkyHeartMeters.calculateHeartRiskAware(
rrintBuffer );
                  }
                  break;

                //...

            } /* end switch on message type */

        } /* end handleMessage() */

    }; /* end Handler */
```

Using Storage Data

Simply use the `calculateHeartRiskAware( String fileName )` method in the NeuroSkyHeart-Meters class:

```
    int heartRiskAwarness = neuroSkyHeartMeters.calculateHeartRiskAware( "john" );
```

Note: The parameter "fileName" in the method is the name of the file that stored calculated heart age.

Results

The return value will be a "Heart Risk Awareness" index that will be one of "0", "1","2" or "3". The following information could be provided by the app to the user based on their "Heart Risk Awareness":

HeartRiskAwareness = 0

Your HRV does not appear to be low at this time. Low HRV has been shown to be related to increased risk of heart attack and mortality. This means your HRV suggests you currently have limited or no risk.

HeartRiskAwareness = 1

Your HRV is a relatively low. Low HRV has been shown to be related to increased risk of heart attack and mortality. It is recommended that you stay active and be careful about what you eat. You could eat foods that can prevent heart attack, such as nuts, fish, coarse grains, vegetables, and you may also drink green tea.

HeartRiskAwareness = 2

Your HRV is low. Low HRV has been shown to be related to increased risk of heart attack and mortality. Bad habits that influence the health of your heart include consumption of food with high fat and sugar content, smoking, drinking alcohol, lack of exercise, high mental pressure, and long periods of sleep deprivation. It is recommended that you change your bad habits by drinking only a moderate amount of alcohol, eating healthy, getting appropriate exercise, controlling your body weight, developing good sleeping habits, and keeping a peaceful state of mind.

HeartRiskAwareness = 3

Your HRV is very low. Low HRV has been shown to be related to increased risk of heart attack and mortality. It is recommended that you change some of your habits. You may consider to quit smoking, stop drinking alcohol. You should also make sure to get appropriate amount of exercise, control your body weight, develop good sleeping habits, eat more fiber and less salt, and keep a peaceful state of mind. Symptoms of heart attack include chest pain, shoulder pain, trouble breathing, poor digestion, and severe fatigue. Please see a doctor if these symptoms occur to you.

(for References, see Heart Age)

HEART_AGE

The Heart Age data value provides an indication of the relative age of a subject heart, based on their Heart Rate Variability (HRV) characteristics as compared to the general population. A low HRV is associated with an increased risk of mortality, and is represented by a Heart Age that is possibly higher than the user's biological age (such as a 35 year old with HRV characteristics that suggest a heart age of 45). The calculation will take into account the user's reported biological age. Use of this data value is only recommended for subjects that are at least 10 years old (biological age).

To receive these values via MSG_HEART_AGE Messages to your app's Handler, first set the user's biological age via the TGDevice object: `tgDevice.inputAge = 25` (of course replacing 25 with the user's actual age). Then, simply have the TGDevice connected to a ThinkGear ECG/EKG sensor (CardioChip), and a user contacting the ECG/EKG sensor hardware properly for at least 60 heart beats continuously with a good, clean signal (SENSOR_STATUS >= 200 for 60 heart beats). If the signal is ever interrupted, and SENSOR_STATUS becomes anything less than 200, then this calculation is reset and starts over, requiring another 60 heart beats of clean data before it can report a MSG_HEART_AGE.

For best results, the user should be sitting calmly during data collection.

An example of how your app and users could potentially use this information would be if your app displayed messages like the following to the user based on their Heart Age value:

Adolescent heart: < 25 years old

Your heart age is xx years old, which is greater/less than your actual age by xx years. Your young heart age allows you to be energetic and to think actively, which helps you deal with demanding work and exercise. A young heart also needs to be taken care of. It is recommended that you avoid staying up late at night, get appropriate amounts of exercise, and maintain a peaceful and positive attitude. You should also eat more fresh fruits and vegetables and cut down on fatty foods to keep your heart at its young state.

Young heart: 26 – 39

Your heart age is xx years old, which is greater/less than your actual age by xx years. You have a mature heart. While in a tense working environment, please don't forget to get good amounts of sleep and exercise, eat well, and take good care of yourself.

Middle-aged heart: 40 – 55

Your heart age is xx years old, which is greater/less than your actual age by xx years. Please pay close attention to your heart health and plan your work and life accordingly to lessen the burden on your heart. It is recommenced that you eat foods that are good for your heart, such as fish, whole grains, beans, nuts, vegetables, red wine, and green tea. You should also get a reasonable amount of exercise to strengthen your heart.

Young elderly heart: 56 – 70

Your heart age is xx years old, which is greater/less than your actual age by xx years. Your heart's function is taking a step towards old age. It is recommended that you live with discipline and avoid straining your body or becoming overly excited or nervous. You should also have regular physical examinations and eat more foods that are good for your heart, such as fish, coarse grains, beans, nuts, vegetables, red wine, and green tea. It is also important for you to get a reasonable amount of exercise so that your heart continues to work effectively.

Elderly heart: >70 years old

Your heart age is xx years old, which is greater/less than your actual age by xx years. It is recommended that you regularly visit your doctor to get physical examinations and carefully follow your doctor's instructions in order to prevent and treat heart disease. You should also get appropriate amounts of exercise and keep a peaceful state of mind. Living with discipline and eating healthy can improve the function of your cardiovascular system and prevent heart disease.

(References)

1. Res Sports Med. 2010 Oct; 18(4):263-9. Age and heart rate variability after soccer games. Yu S, Katoh T, Makino H, Mimuno S, Sato S.

2. J Am Coll Cardiol. 1998 Mar 1; 31(3): 593-601. Twenty four hour time domain heart rate variability and heart rate: relations to age and gender over nine decades. Umetani K, Singer DH, McCraty R, Atkinson M.

3. Am J Cardiol. 2010 Apr 15; 105(8): 1181-5. Epub 2010. Relation of high heart rate variability to healthy longevity. Zulfiqar U, Jurivich DA, Gao W, Singer DH.

4. Cardiovasc Electrophysiol. 2003 Aug; 14(8): 791-9. Circadian profile of cardiac autonomic nervous modulation in healthy subjects: differing effects of aging and gender on heart rate variability. Bonnemeier H, Richardt G, Potratz J, Wiegand UK, Brandes A, Kluge N, Katus HA.

5. Pacing Clin Electrophysiol. 1996 Nov; 19(11 Pt 2): 1863-6. Changes in heart rate variability with age. Reardon M, Malik M.

Personalization

This algorithm allows the Connector to try to recognize a connected user based on their ECG/EKG data. To use it, one or more users should "train" their ECG/EKG data into the Connector. Then, whenever the Connector is reading ECG data from a user, it can attempt to identify which of the trained users (if any) is the one it is reading from.

There are two steps to use the Personalization algorithm:

Training

The first part records ECG/EKG data of user, using the `EKGstartLongTraining( String userName )` method in the Connector class. If the user then keeps good, clean contact with the ECG/EKG sensor hardware properly, a `MSG_EKG_TRAIN_STEP` Message will be sent to your app's Handler to show which step you are currently on. After two steps are finished, it will send a `MSG_EKG_TRAINED` Message to your app's Handler to indicate that the recording is finished.

```
   connector.EKGstartLongTraining("NeuroSky");
```

Detection

This part is used to recognize user based on saved data from the first part. To recognize user, invoke the `EKGstartDetection()` method. Then, if the user keeps a good, clean contact with the ECG/EKG sensor hardware, the Connector will send a `MSG_EKG_IDENTIFIED` Message to your app's Handler. The return value will be one of the registered user names, or "Unknown".

```
  connector.EKGstartDetection();
```

Event Handler for EKGPersonalizationEvent

The EKGPersonalizationEvent event handler indicates which state the personalization algorithm is currently in. There are four possible states:

1. `MSG_EKG_IDENTIFIED` indicates that the algorithm has determined who the user is. The name will be returned in the `dataMessage`.
2. `MSG_EKG_TRAINED` indicates that the final training step is complete. Typically, the next step would be to call `EKGstartLongTraining`.
3. `MSG_EKG_TRAIN_STEP` indicates that a training step has been completed. The number of training steps completed is returned in the `dataMessage`.
4. `MSG_EKG_TRAIN_TOUCH` indicates that the user should now make good, clean contact with the ECG/EKG sensor hardware.

In the event handler for EKGPersonalizationEvent, do the following:

```
static void OnEKGPersonalizationEvent(object sender, EventArgs e) {
    EKGPersonalizationEventArgs ekgArgs = (EKGPersonalizationEventArgs)(e);
    int status = ekgArgs.statusMessage;

    switch(status) {
        case 268:
            string data = (string)(ekgArgs.dataMessage);
            Console.WriteLine("status = MSG_EKG_IDENTIFIED " + " and username = " + data);
            break;

        case 269:
            Console.WriteLine("status = MSG_EKG_TRAINED");
            break;

        case 270:
            int trainStep = (int)ekgArgs.dataMessage;
            Console.WriteLine("status = MSG_EKG_TRAIN_STEP " + " and training step = " + trainStep
+ ". Please remove fingers from sensors");
            break;
        case 271:
            Console.WriteLine("status = MSG_EKG_TRAIN_TOUCH.  Please place fingers on sensors");
            break;
    }
}
```

To use this event handler, add following code in your main function:

```
Connector.EKGPersonalizationEvent += new EventHandler(OnEKGPersonalizationEvent);
```

Important: Note that Connector.EKGPersonalizationEvent is a static event handler.

RrInt

Whenever an R-peak is detected along a user's PQRST ECG/EKG, then a `RrInt` data type is sent to your app's data event handler indicating the R-R interval, in milliseconds, since the last R-peak.

Proper App Design

Important: 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

Note: There are currently no known issues. If you encounter any bugs or issues, please visit http://support.neurosky.com, or contact 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.

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.

All ECG data should be collected while the user is seated quietly, breathing regularly, with minimal movement, for best results.

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.