ThinkGear SDK for .NET: Development Guide and API Reference

December 20, 2012

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)
- ThinkGear.dll library
- supporting libraries: NLlog.dll/config/xml, JayrockJson.dll
- HelloEEG Sample Project
- TG-HelloEEG.exe a reference build of the HelloEEG sample project

You'll find the "ThinkGear.dll" in the `libs/` folder, and 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

- BrainAthlete
- MindTune
- 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("MindWandering") ) {
            Console.WriteLine( "MindWandering Level:" + tgParser.ParsedData[i]["MindWandering"] );
        }
    }
}
```

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

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.

TGParser Class

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
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
MindWandering	Mind Wandering Level. The Mind Wandering Level ranges from 1 (low Mind Wandering) to 10 (high Mind Wandering). The Mind Wandering algorithm updates once every 0.5 seconds, assuming PoorSignal is less than 51. If PoorSignal is above 51, nothing is returned.	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
Raw	EEG	short
HeartRate	Heart rate (BPM)	double
RrInt	Time between detected heart beats in milliseconds	uint

Note: As each packet is received in the ThinkGear DLL, the associated timestamp is also recorded