Documentation on the BrwExtReader.dll

Version 1.0.1

Date Jan 2012

3Brain GmbH Schulstrasse 76 7302 Landquart Switzerland
Tel +41 813227086 Fax +41 813227087 info@3brain.com 3brain.com

Contents

1. Introduction

1.1. About this manual

This manual comprises information about the BrwExtReader library that can be used to read BrainWave data files (BRW-file). It is assumed that the reader already has a basic understanding of technical and software terms.

BrwExtReader.dll has been written in the C# programming language under the .NET framework.

BrwExtReader.dll can be used to implement a custom application in order to access offline data recorded with the BioCAM acquisition systems.

1.2. Requirements

The BrwExtReader.dll library can be used on managed languages under Windows platforms. Tests under Mono and other operating systems have not been made. The .NET framework 3.0 or higher needs to be installed. The library can also be accessed from MATLAB starting with release 2009a.

2. The BRW-file

The BRW-file contains data written with the BrainWave software tool and acquired from the BioCAM system. The file allows managing and storing different types of information both on the experiment session and on the recorded signals.

With the BrwExtReader library the user can have access to basic information on the experiment and to the data that has been recorded. In particular, the user can read both the raw data acquired from the microelectrode array (MEA) mounted on the BioCAM system and the spike data obtained with real-time analysis.

Raw and spike data represent different streams within the BRW-file that can be associated to specific array positions. A MEA array position (electrode/channel) is identified through two coordinates, namely the row and the column coordinate of the channel within the MEA. A BRW-file is made up of at least one stream of data (raw or spike) and may also contain multiple streams for the same electrodes, depending on the acquisition settings.

Raw data is acquired online by sampling the voltage signal sensed at each electrode site (or at a subset of electrodes if otherwise specified by the user).Transmission and storage of the full-array activity is performed as a sequence of images (frames). Single microelectrode raw data can then be reconstructed by recombining single electrode data from sequential frames. The BrwExtReader library can perform this reconstruction and allow the user to load from the BRW-file the raw data of one or more electrodes for a specific time range.

Spike data is derived by processing online the incoming raw data with spike detection algorithms that analyze the entire array (or a channel subset if otherwise specified by the user) and store on the BRW-file spatial and temporal information on the detected spikes. The BrwExtReader library is able to read the spike stream and return the spike information within a specific spatial and time range.

3. Using the library in .NET

The user can find detailed information on all the functionalities of the library within the help documentation that is attached to the library. In what follows the basic functions used to load data from a BRW-file are discussed. Usage examples are written in C# and to use them in an application the user needs to have a reference to the BW namespace that contains the classes that will be described later. Under Visual Studio the reference to the BrwExtReader.dll must be added by clicking on Project->Add Reference… and locating the library. The following command at the beginning of each source file has to be added:

```
using BW;
```

3.1. The ChCoord class

The ChCoord class is used to represent electrodes of the MEA device. A channel is identified by its spatial coordinates, i.e. from the row and the column of its position.

The following example shows some functionalities of the ChCoord class:

```
// print the row and column coordinates
ChCoord ch = new ChCoord(35, 55);
Console.WriteLine("The channel has coordinates: row: {0}; column: {1}", ch.Row, ch.Col);

// initialize an array of channel coordinates.
ChCoord[] chs = new ChCoord[] { new ChCoord(3, 10), new ChCoord(50, 5), new ChCoord(20, 10)
};

// default sorting is made by row, i.e. a channnel precedes another if its row coordinate is
// smaller and within the same row if its column coordinate is smaller
Array.Sort(chs);
Console.WriteLine("Channels sorted in ascending order using the default sorting
method\n{0}\n{1}\n{2}", chs[0], chs[1], chs[2]);

// sort by looking firstly to columns
Array.Sort(chs, new ChCoord.ByColumnComparer());
Console.WriteLine("\nChannels sorted in ascending order using the comparer that sort by
column\n{0}\n{1}\n{2}", chs[0], chs[1], chs[2]);

Console.WriteLine("\nClick return to exit...");
Console.ReadLine();
```

3.2. The BrwRdr class

The BrwRdr class is used to read the BRW-file. Once a new instance of the class has been initialized, the Open method is called to load a BRW-file. Close or Open are called to unload the opened file or to switch to another file.

```
// Initialize an instance of the BrwRdr class
BrwRdr brwRdr = new BrwRdr();
// you must use a valid full file path
brwRdr.Open("yourBrw.brw");
```

The RecDuration property is called to know the duration in number of milliseconds of the acquisition contained in the BRW-file, while the RecNFrames property returns the duration in number of frames. The SamplingRate property returns the sampling frequency in Hz of the recording.

```
// the number of frames of the recording
long nFrames = brwRdr.RecNFrames;
// the duration in seconds of the recording
double nSec = brwRdr.RecDuration / 1000;

// the sampling frequency
int sf = brwRdr.SamplingRate;
```

The NStream property returns the number of streams that have been stored into the BRW-file and the StreamTypes property returns the array of stored StreamType, where StreamType is an enumerator for all the possible stream that can be contained in a BRW-file.

ExistRawStream and ExistSpikeStream methods are used to know which type of streams are contained in the BRW-file. The user can then get the number and the coordinates of the channels that are available for a specific stream by calling the GetNRecChs and GetRecChs methods respectively. To check whether a certain stream has been recorded for a a specific channel IsRawStreamFor and IsSpikeStreamFor methods can be called. The GetRecChsUnion method returns the array of channel coordinates that have been recorded overall, i.e. the union of the channels that have been recorded for each existing stream. The GetRecChsIntersection method returns instead the intersection of the channels that have been recorded for each existing stream.

To load raw data you can use both the GetRAwData and GetRawDataADCCounts methods, where the former returns values converted in microVolt and the latter returns values in ADC counts. The GetRawData methods can be used by specifying the channel(s) coordinates and the frame range to load. To access spike data, GetSpikes methods return the spike timestamps in number of frames. GetNDetectedSpikes might also be used to know in advance the number of detected spikes for specific channel(s).

In the following example, the mean firing rate for each spike recorded channel is computed:

```
// check whether a spike stream exist
if (brwRdr.ExistSpikeStream())
{
       // gets the channels with a spike stream
       ChCoord[] spikeChs = brwRdr.GetRecChs(StreamType.Spikes);
       // the number of detected spikes for each channel
       int[] nSpikes = brwRdr.GetNDetectedSpikes(spikeChs);

       // the mean firing rate for each channel
       double[] mfr = new double[spikeChs.Length];
       // the overall mean firing rate
       double overallMfr = 0;

       for (int i = 0; i < spikeChs.Length; i++)
       {
             mfr[i] = nSpikes[i] / nSec;
             overallMfr += mfr[i];
       }
       overallMfr /= spikeChs.Length;

       Console.WriteLine("Overall mean firing rate: {0} spike/sec", overallMfr);
}
```

4. Using the library from Matlab

4.1. Overview

MATLAB 2009a or higher allows to manage .NET assemblies (for detailed information the user can reference to this online resource). Nonetheless there are some limitations in the interoperability between .NET and MATLAB that will be discussed later on.

MATLAB can load a private .NET assembly, i.e. a library that is located in a certain specific folder, and access the classes defined into the assembly. The command allowing for adding the assembly is:

```
asm = NET.addAssembly('fullPath');
```

where the full path is the path of the BrwExtReader.dll comprising the extension. The user can then explore the asm variable in the MATLAB Variable Editor to view the components (classes, enums, etc.) that he can access. Once the assembly is loaded it cannot be unloaded unless by quitting MATLAB.

From MATLAB the two classes, i.e. the ChCoord class and the BrwRdr class, that are needed to load data from a BRW-file, are visible under the BW namespace. Instances of these classes appear in MATLAB as reference types, i.e. they are only handles to the underlying data and copying or passing these instances to a method equals to copy or pass only the handle. To create an instance of the BrwRdr class, a reference to it by its qualified name, including also the namespace, is required:

```
brwRdr = BW.BrwRdr;
```

4.2. Limitations

A complete list of limitations to the MATLAB .NET support can be found here.

For what regards the matter of the present manual, the main limitations that the user should consider are: (i) MATLAB cannot create or perform some operations with jagged arrays and (ii) MATLAB indexing of .NET arrays has some restrictions.

A jagged (or ragged) array is a non-rectangular array that differs from normal multi-dimensional arrays (for 2-dimensional array: a matrix [,] in MATLAB or for instance a System.Double[,] under .NET) since is an array whose elements are arrays (for a 2-level jagged array of doubles the notation under .NET is System.Double[][]). The elements (sub-arrays) of a jagged array can be of different sizes and this type of array is used in the BrwExtReader library.

The reason for using jagged arrays resides firstly in the faster management of such arrays with respect to multi-dimensional arrays under the .NET environment. Moreover, in some cases non-rectangular arrays are more convenient. For instance when willing to load for two electrodes the spikes that have been detected within a certain time range and that may differ in number between the two channels.

The user cannot directly convert a jagged .NET array into a MATLAB array as he would do for a multi-dimensional .NET array by simply applying the double operator. However, the user can access the underlying arrays and convert them. For instance, giving that data is a MATLAB handle to a jagged array of type System.Double[][], the programmer can generate a MATLAB array for the nested array at position 0 into the jagged array by typing:

```
a = double(data(ff(1));
```

For what concerns array indexing, the user should consider that only scalar indexing is supported for accessing .NET arrays' elements, being the MATLAB colon operator not supported. Hence, when you want to derive an array whose elements are a subset of an existing .NET array, the user needs to create a .NET array from MATLAB and then to assign the subset elements to the new created array. Supposing for instance to have a chs MATLAB variable as a handle to an array of BW.ChCoord made up of 100 elements and to want to derive another .NET array made up of only the first 10 elements, the user might type:

```
chsSubset = NET.createArray('BW.ChCoord', [10]);
for i = 1 : 10
      chsSubset(i) = chs(i)
end
```

Another aspect that has to be considered when working in MATLAB with .NET assemblies is how types are returned when calling members of .NET classes. When the user returns a value from a .NET method, in some cases MATLAB leaves it in the original format, in others it does not. MATLAB keeps the original format in case the type has not a direct MATLAB type counterpart (e.g., an instance of the ChCoord class) or in case the value is array. By having the value in the original .NET format the user can keep on interacting with other .NET methods (for instance passing a ChCoord to the BrwRdr.IsRawStreamFor method).

When original .NET types are automatically converted in MATLAB types the user has to take care of the conversion that toake place. For instance, calling the property in the BrwRdr class that returns the sampling frequency:

```
sf = brwRdr.SamplingRate;
```

returns an Int64 sf value (automatically transformed from System.Int64 .NET type to an Int64 MATLAB type). The returned value will limit the use of it and for instance the following operation:

```
x = (1 : 100) / sf;
```

will produce an error since integer are not supported in MATLAB for division operations. Hence a good practice is to convert the values that are returned to the default MATLAB type, i.e. the double type, by:

```
sf = double(brwRdr.SamplingRate);
```

4.3. Example

In what follows the example shown before for a .NET application converted in MATLAB looks like:

```
% load the private assembly by the full file path
asm = NET.addAssembly('BrwExtReaderfilePath');

% create an instance of the BrwFile class
brwRdr = BW.BrwRdr;

% open a BRW-file (the returned value indicates whether the opening was
% successful)
succ = brwRdr.Open('yourBrw.brw');

if succ
     try
        % the duration in seconds of the recording
        nSec = brwRdr.RecDuration / 1000;
        % check whether a spike stream exist
        if (brwRdr.ExistSpikeStream())

            % gets the channels with a spike stream
            spikeChs = brwRdr.GetRecChs(BW.StreamType.Spikes);
            % the number of detected spikes for each channel
            nSpikes = double(brwRdr.GetNDetectedSpikes(spikeChs));

            % the mean firing rate for each channel
            mfr = zeros(spikeChs.Length, 1);
            % the overall mean firing rate
            overallMfr = 0;

            for i = 1 : spikeChs.Length
                mfr(i) = nSpikes(i) / nSec;
                overallMfr = overallMfr + mfr(i);
            end
            overallMfr = overallMfr / double(spikeChs.Length);

            msgbox(strcat('Overall mean firing rate: ', ...
               num2str(overallMfr), 'spike/sec'));
        end
     catch ex
        brwRdr.Close();
        throw(ex)
     end
end
```