Behavioral Neuroscience Program Brain Computer Interface Lab Manual: *The SpikerBCI*

# Introduction

# Installing SpikerBCI

# Getting Started

This section provides brief step by step instructions for operating the SpikerBCI. Additional detail is provided in the Operating the SpikerBCI section.

## Starting the BNS BCI controller

1. **Attach the Spikerbox to the host PC.** Connect the mini-USB into the Spikerbox and the USB to the host PC.
2. **Start MATLAB on the host PC**. You should be using MATLAB version 2022 or higher.
3. **Type BYB\_BCI in the MATLAB command window**.

## Operating the BCI

1. **Select the communications port for the Spikerbox.** This should be automatic if the Spikerbox was plugged in before BYB\_BCI was started. Left click the Configure->Port menu. A list of available Ports will be presented. Select the port to which the Spikerbox is attached.
2. **Set the input buffer length.** The default is 0.2 seconds. Select Configure->Buffer Length. Select a buffer length option.
3. **Load a data handler.** Left Click on the Handler option and select Load. A dialog box will appear showing the contents of the Handler folder. Select the desired handler and left click Open.
4. **Turn the Spikerbox on.**
5. **Initialize the handler.** Left click the Initialize button.
6. **Start the BCI.** Left click the Start button.

## Loading a new handler

1. **Stop the BCI.** Left click the Stop Button.
2. **Load a data handler.** Left Click on the Handler option and select Load. A dialog box will appear showing the contents of the Handler folder. Select the desired handler and left click Open.
3. **Initialize the handler.** Left click the Initialize button.
4. **Start the BCI.** Left click the Start button.

# Operating SpikerBCI

Graphical user interface, application, chat or text message

Description automatically generated

# Creating Your Own Data Handler

## What is a data handler?

Data handlers are MATLAB scripts that determine how the SpikerBCI *handles* and displays data. A data handler has two roles. The first is to initialize any processing object and display elements prior to the onset of data collection. This occurs when the user clicks the Initialize button on the main SpikerBCI interface. The second is to perform the real-time processing of the incoming data from the Spikerbox. This happens each time SpikerBCI receives a data packet the length of which is determined by the Buffer Length parameter set in the main SpikerBCI interface. Using the default value of 0.2 second buffer length, the handler will be called 5 times per second.

NOTE: The convenience offered by a high level programming language like MATLAB also make it quite slow for some kinds of operations (e.g. plotting) so you should be careful about how much processing and plotting you include in your handler. If the handler cannot keep up with the data rate determined by the buffer length there will be a growing lag between input to the Spikerbox and output from your handler.

## The parts of a handler

A handler must be comprised of three functions. An entry function, an initialize function and an analyze function. Although these functions do not need to embedded in a single script file, it is convenient to do so.

### The entry function

The entry function (lines 1 to 7 in the example below) is the function that is called by SpikerBCI and acts as the entry point into the handler and by MATLAB convention, should have the same name as the handler. This purpose of this function is to route calls to either the initialize or analyze functions based on whether data was passed to the handler. If using the template created by SpikerBCI when creating a new handler, this function should not be edited. The entry function takes two arguments (inStruct and varargin) and returns one argument (outStruct).

***inStruct*** is a MATLAB structure (the name is short for input structure) that holds parameters including variables and objects that that the BCI needs to access for proper function. Any information that should be remembered from over time (e.g. across multiple calls to your handler) should be added to this structure. I will add a complete list of variables in inStruct in time.

***varargin*** is a MATLAB input variable that allows the function to handle a variable number of inputs. During operation of the BCI, this variable will hold the vector of voltage values measured from the input to the Spikerbox and a vector of event values indicating the status of the digital pins on the Spikerbox.

***outStruct*** (short for output structure) is a simply copy of *inStruct* that includes any changes made by the initialize and analyze function. This is the version that will be passed as *inStruct* the next time the handler is called.

### The initialize function

The initialize function is called when the user clicks on the Initialize button on the SpikerBCI interface. The initialize function should contain any code that needs to be run only once prior to real-time operation of your BCI. This would include initializing displays, creating analysis objects (see the Extensions section), creating and opening files for saving, etc.

The initialize function returns the input variable back to its calling function so that any changes made to *p* will be stored.

### The analyze function

The analyze function is called whenever the SpikerBCI receives a complete data packet and should contain code that you want to run on each data packet. The function must accept 3 input variables. The first is the *inStruct* variable described in the entry function section. The second is a vector of double values with length equal to the number of samples in your input buffer. This is the digitized input to the Spikerbox. The third is a vector of unsigned 8 bit integers with length equal to the number of samples in your input buffer. This is the 2-bit event maker read from the digital pins on the Spikerbox.

## An example handler

The section below contains the code from the *singleChart* handler. This handler uses the BYB\_Chart object to crate and update a simple plot of the incoming digitized data in real-time. I will describe each function in brief to give some concept of what each is doing. Note, however, that the goal of this section is not to teach you how to code in MATLAB.

The complete entry function looks as follows. Note that the line numbers are added for convenience but do not appear in MATLAB.

function outStruct = singleChart(inStruct, varargin)

if nargin == 1

outStruct = initialize(inStruct);

else

outStruct = analyze(inStruct, varargin{1}, varargin{2});

end

end

First notice that the entry function has the same name as the script file (singleChart). All this function is doing is checking to see how many variables have been passed to the function (stored in *nargin*). Only the *inStruct* variable is passed to the handler during initialization (there is no data so it is not passed to the function) and because there is only one input (*nargin == 1*), the initialize function is called. Otherwise, the analyze function is called. Note that *inStruct* is passed to both function and *outStruct* is received from both functions before being return from the entry function. The digitized data (*varargin{1}*) and the event marker channel (*varargin{2}*) are passed separately to the analyze function.

on that will call the handler, I thought it was OK.

The entire initialize function from the simpleChart example is as follows:

function p = initialize(p)

existingFigure = findobj('Name', 'Very Simple BYB BCI Data Display');

if ~isempty(existingFigure)

p.handles.outputFigure = existingFigure(1);

clf(p.handles.outputFigure);

else

%create a new figure to hold all the plots etc

p.handles.outputFigure = figure('Position',[200,200,1000,300]);

%name it so we can recognize it later if the software is rerun

p.handles.outputFigure.Name = 'Very Simple BYB BCI Data Display';

end

ax = axes(p.handles.outputFigure);

p.Chart = BYB\_Chart(p.sampleRate,5, ax);

end

Note that the initialize function is actually named *initialize.* This is not necessary but is the convetion used in the template an good practice. If you change the name of the initialize function, make sure you change the corresponding name in the entry function.

The input to the initialize function is the *inStruct* variable described in the entry function, however, here it has been named *p* which is short for parameters. The shorter name and helps to reduce the amount of typing required.

In the simpleChart example the initialize function is creating a MATLAB figure window and a plotting axis on that figure so that the analyze function will have a place to plot data. It is also initializing an instance of a BYB\_Chart object that will be used to plot the incoming data.

To avoid creating a new figure each time the user clicks the Initialize button on the main SpikerBCI interface, the first *findobj* function is checking to see if the figure we want to create already exists. If it does exist we assign the figure to a variable called outputFigure that we embed in a handles structure which is part of the p structure. We also clear any contents on the figure.If it does not exist we create it with the *figure* function and assign a name to the figure. We use the assigned name to check to see if the figure exists so the name in line 3 should match the name in line 11.

Once we have an empty figure, we create an axis called *ax* on that figure using the *axes* function (line 14).

We then create a BYB\_Chart object (line 15) called *Chart*. Note that *Chart* is added as an item in the structure *p* by assigning the name *p.Chart*. This is critical to ensure that the chart object is available in future calls to the analyze function. The chart object needs to know the sample rate (already stored in the parameter structure *p)* the number of seconds of data to display on the chart (5 seconds in the example) and where to plot the data (the axis we just created).

You may have noticed that we save the *Chart* object in our parameters structure *p*, but we did not store our plotting axis called *ax* even though we need to access it later when plotting data. This is because the plotting axis is stored internally in the *Chart* object so there is no need to store it separately if only the *Chart* object is going to use it.

The analyze function from the simpleChart handler is as follows:

function p = analyze(p,data, event)

p.Chart = p.Chart.UpdateChart(data, event, [0, 1.3]);

end

Yup, that is really all there is. The function accepts the sampe parameter structure described in the entry function and initialize function (*p*), as well as the data and event vectors described above. Aside from the function definition, there is only a single line of code that calls the *Chart* object we created in our initializer. We are calling the BYB\_Chart method *UpdateChart* to add the current data packet to the existing plot. The data and event vectors are passed to the chart and a plot range is also passed so the chart scale will always be between 0 and 1.3 mV.

A couple important things to note are that, the BYB\_Chart object we created (called *Chart*) is stored in the p structure so we have to call it is *p.Chart* and that the BYB\_Chart returns an updated copy of itself when called so it is important to assign the Chart variable to its own output. Hence the *p.Chart = p.Chart.UpdateChart*. If you did not so this, the new data you just added would not be remembered the next time around.

The code snippets above describe the individual functions contained in the singleChart.m handler. For reference, the entire singleChart.m Handler file is organized as follows.

function outStruct = singleChart(inStruct, varargin)

if nargin == 1

outStruct = initialize(inStruct);

else

outStruct = analyze(inStruct, varargin{1}, varargin{2});

end

end

%this function gets called when data is passed to the handler

function p = analyze(p,data, event)

p.Chart = p.Chart.UpdateChart(data, event, [0, 1.3]);

end

%this function gets called when the analyse process is initialized

function p = initialize(p)

existingFigure = findobj('Name', 'Very Simple BYB BCI Data Display');

if ~isempty(existingFigure)

p.handles.outputFigure = existingFigure(1);

clf(p.handles.outputFigure);

else

%create a new figure to hold all the plots etc

p.handles.outputFigure = figure('Position',[200,200,1000,300]);

%name it so we can recognize it later if the software is rerun

p.handles.outputFigure.Name = 'Very Simple BYB BCI Data Display';

end

ax = axes(p.handles.outputFigure);

p.Chart = BYB\_Chart(p.sampleRate,5, ax);

end

## Creating a new handler

There are two primary ways to create a new handler. You should choose the option that works best for your needs and your programming experience.

1. Start with an existing handler. The Handlers folder contains several example handlers that demonstrate different aspects of processing and plotting data. You can create a copy and edit the handler that is closest to your needs.
2. Create a blank handler from a template. To create an empty handler from a template, click on the Handler menu in the SpikerBCI interface and select New. You will be prompted to enter a name for you new handler which will be added to the Handler folder and then opened in MATLAB.

# SpikerBCI Extensions

SpikerBCI extensions are MATLAB objects that can be used to display and analyze data from the Spikerbox. They are designed to provide common functionality to the user, but require minimum coding to implement. All extensions can be found in the Extensions subfolder and you should be able to access detailed help about each extensions by type

help extension

in the MATLAB command window, where extension is the name of the extension.

Extensions are just MATLAB objects so if you know about object-oriented programming in MATLAB (or more generally) you will have a good idea of how extensions work. For those who are no familiar with object-oriented programming, extensions have properties and methods. Properties are information that the extension stores and may occasionally update and methods are operations that the extension performs.

To help explain SpikerBCI extensions let’s look at a simple fictitious example of a MATLAB object that represented information about a bank account. The object might contain a starting balance to indicate how much money initially added to the account and a current balance to indicate what the current balance is. The properties of an object are first set when it is created so lets see what creating this object might look like,

myAcconut = BankAccount(1000);

We have created an instance of BankAccount called myAccout with an initial balance of 1000. Lets assume because we are just creating this object that 1000 is used for both the Initial and Current balance. So to see the current balance you might type

myAccount.CurrentBalance

to access information about the current balance, after which you would see

ans =

1000

in the matlab window showing us that the current balance is 1000. Now lets assume there is a spend method that subtracts the amount of money you have spend from the current balance. To indicate that we spent 100 dollars we would pass the amount spent to Spend method like this:

myAccount = myAccount.Spend(100);

Assuming that our spend methods simply subtaracts the amount spent, if you query the current balance you should see the change.

myAccount.CurrentBalance

ans =

900

Note that we assign the output of myAccount to itself. This is because myAccount will return an updated copy of itself with the new information. If we do not make this assignment the new interval value of CurrentBalance will not be saved.

So that is a brief introduction into objects, and as I said earlier, SpikerBCI extensions are just objects to they work the same way. In the next section I provide information about the different properties and methods of each extension.

# Extension Reference

Only a brief description is provided here. For full details type

help extension

in the MATLAB command window where extension is the name of the extension you want more information about.

HBSpikerBox: The HBSPikerBox object controls communication with and acquisition from the Back Yard Brains Heart Brain SpikerBox. This object will establish communication with the Spikerbox over the serial port, acquire data, store it in a circular buffer (at least is will eventually

see BYB\_CircularBuffer) and return data frames to a user defined handler for further processing.

*The HBSpikerBox extension is implemented in the SpikerBCI and is not required unless you are creating your own BCI interface.*

SBCI\_Chart: Initializes and updates a scrolling chart object to eiwplqy a predefined duration of data that extends beyond the duration of the input buffer. For example, the SBCI\_Chart extension allows for plotting 5 the last 5 seconds of data that is arriving to the BCI in .2 second intervals.

# Appendix I. The Spikerbox

The Back Yard Brains(BYB) Heart and Brain Spikerbox is built on an Arduino platform which uses a 10-bit analog to digital converter (ADC) and returns values in the range from 0 to 1023 (2^10-1). Because the reference signal on the Arduino is 0 to 5V, the resolution of the ADC is 4.88 mV (5V/1023 = .00488V). According to the circuit diagram on the BYB website, the analog signal is amplified with a gain of 3840 (three stages of 80X, 32X, and 1.5X) which translates to an input range of 0 to 1.3 mV (5V/3840 = .0013V). To center the analog signal (and allow for both positive and negative values) in the input range, the Spikerbox adds a DC offset of 2.5V prior to which make the actual input range ± 0.65 mV or ± 650 uV and the resolution is 1.27 uV.

# Appendix II. The data frame

The firmware on the Spikerbox samples the analog and digital inputs to the Spikerbox continuously at 1000 Hz. The analog input is the biosensor channel and contains the difference between the two input electrodes. The digital input is read from pins9 and 11 on the Spikerbox expansion port. Unless something is connected to these pins, they will likely float high, but can fluxuate.

For a single sample, the event markers and digitized data are combined into a single two-byte (16 bit) data frame. The SpikerBCI HBSpikerBox object will automatically decode the input signal and apply a multiplier to convert from AD values to millivolts. However, if you want to write custom MATLAB code to decode the data, you can use the information presented in this section.

The most significant bit of the first byte will always be 1 and can be used to identify the frame onset. The event marker information is stored in bytes 6 (E1 below from pin 9) and 7 (E2 below from pin 11) of the first byte. The 10-bit data sample is stored across the 2-byte frame. The 3 least significant bits of the byte 1 contains the 3 most significant bits of the 10-bit ADC value (D8-D10). The 7 lowest bits of the byte 2 contain the 7 least significant bits of the 10-bit ADC value (bits D1-D7 below).

A 2-Byte data frame

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | E2 | E1 |  |  | D10 | D9 | D8 |  | D7 | D6 | D5 | D4 | D3 | D2 | D1 |
| MSB | Byte 1 | | | | | | LSB | MSB | Byte 2 | | | | | | LSB |

The following table describes the relationship between the value of the event marker and the state of the digital pins on the Arduino.

|  |  |  |
| --- | --- | --- |
| **Value** | **Pin 11** | **Pin 9** |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

Because each data frame has 2 bytes, the number of bytes in the input buffer should be the number of samples you want to collect x 2. To determine the length of the input buffer use the following formula.

*Fs \* d \* 2*,

Where *Fs* is the sample frequency and *d* is the duration of the buffer in seconds. The sample rate is always 1000 Hz, so for a 25 millisecond data sample, the length of the input butter would be

1000 \* .025 \* 2 = 50 bytes.