NUI Chapter 10. Reading Your Mind with the MindWave

The NeuroSky MindWave headset, modeled by yours truly in Figure 1, is a low-cost electroencephalography (EEG) device (about $100; details at http://www.neurosky.com). It sends its brainwave data to a wireless-enabled USB dongle attached to your PC.

Figure 1. The MindWave is Powerful in that One (or Maybe Not).

This chapter explains how to access the brainwave data in a Java application, including how to display the most relevant information graphically (as in Figure 2).

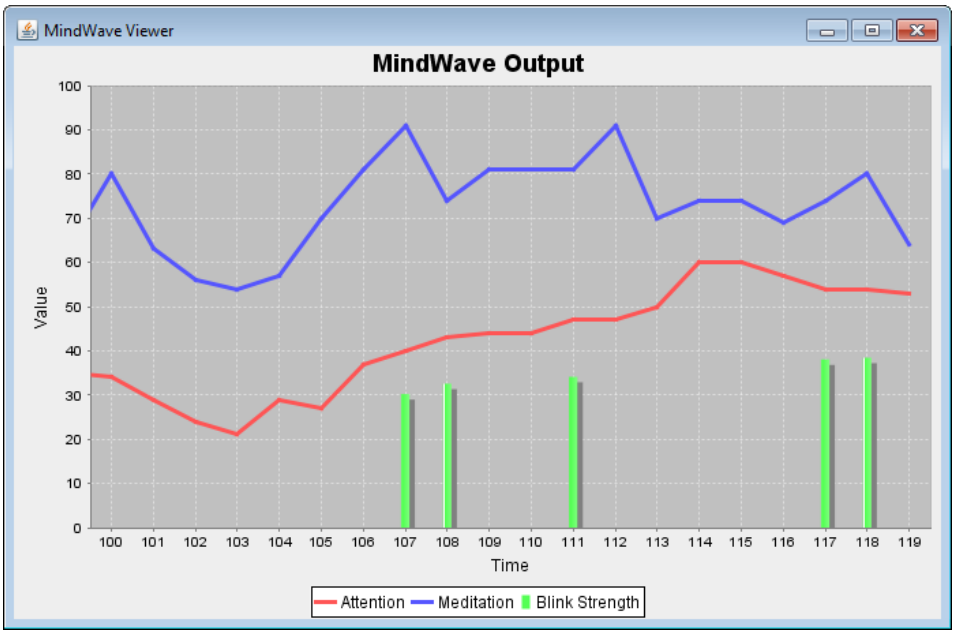


Figure 2. The MindWave Brainwave Chart.

I'll also describe how to create and read a brainwave log file.

The MindWave thankfully dispenses with the traditional hair-net of sensors and conductive jelly, instead utilizing a single electrode pressed against your forehead,

© Andrew Davison 2012

and reference electrodes attached to your left earlobe via a clip. This makes the device comfortable and easy to use, at the expense of some accuracy.

The headset's ThinkGear chip amplifies the raw brainwave signals, removes noise, and outputs proprietary 'attention' and 'meditation' levels (ranging between 0 and 100) along with eye blink strengths (values in the range 0 to 255). According to NeuroSky, the attention value is an indication of the user's level of mental focus, while meditation measures mental calmness. A level of around 50 means 'neutral' or average, with 60 – 80 being 'slightly elevated' and 80 – 100 'elevated'. The levels are updated once per second, and blink strength whenever a blink is detected. The chart in Figure 2 plots the two levels as lines, and blink strength as bars.

It's possible to access more common EEG frequency data: delta, theta, alpha, beta and gamma brainwaves, which are also updated every second, and to read the raw data coming from the MindWave, at a rate of about 512 packets/second.

Probably the biggest drawback of the MindWave is the typical user's overly optimistic expectations about what it can do. Visions of Yoda-style feats of mental dexterity are rapidly deflated when you actually try out the example applications and games.

A large part of the problem is that there's no obvious direct physical means of affecting the attention and meditation values. For instance, the NeuroSky documentation suggests staring at something to increase attentiveness, and closing your eyes to promote meditation. As you might expect, this form of interaction is a lot less predictable and responsive than simply pressing a key or clicking a mouse button.

1. Processing Brainwaves

The MindWave data arriving at the PC can be accessed using the ThinkGear Communications driver (TGCD), which presents it as a stream of data packets arriving at a serial port. Another approach is to invoke the ThinkGear Connector (TGC) daemon, which acts as a TCP server. A client connects to its socket at localhost, port 13854, to read the data packets raw, or in the form of structured JSON tuples.

TGCD and TGC are part of the MindSet Development Tools (MDT), a free download available from http://store.neurosky.com/products/mindset-development-tools. I'll be using the Windows version of TGCD in this chapter, but there's equivalent software for the Mac, Linux, Android, iPhone, and other platforms. NeuroSky hosts a useful development site at http://developer.neurosky.com/, which includes a wiki and forum.

If you're interested in exploring the socket programming option (i.e. the TGC), the details are nicely explained at several websites, including http://eric-blue.com/2011/07/13/neurosky-brainwave-visualizer/, http://crea.tion.to/processing/thinkgear-java-socket,and http://jorgecardoso.eu/processing/MindSetProcessing/.

Figure 3 shows that the ThinkGear Communications driver (TGCD) on Windows consists of a DLL (thinkGear.dll) and my Java interface implemented using JNA (Java Native Access).

 © Andrew Davison 2012

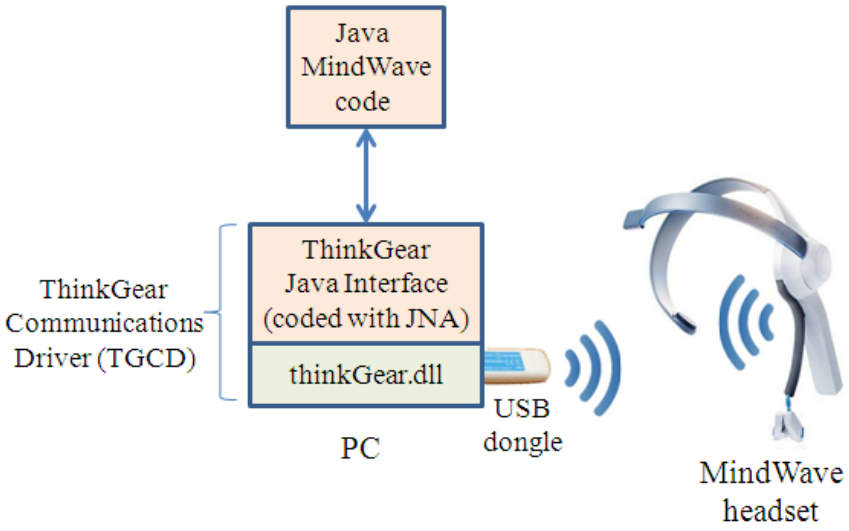
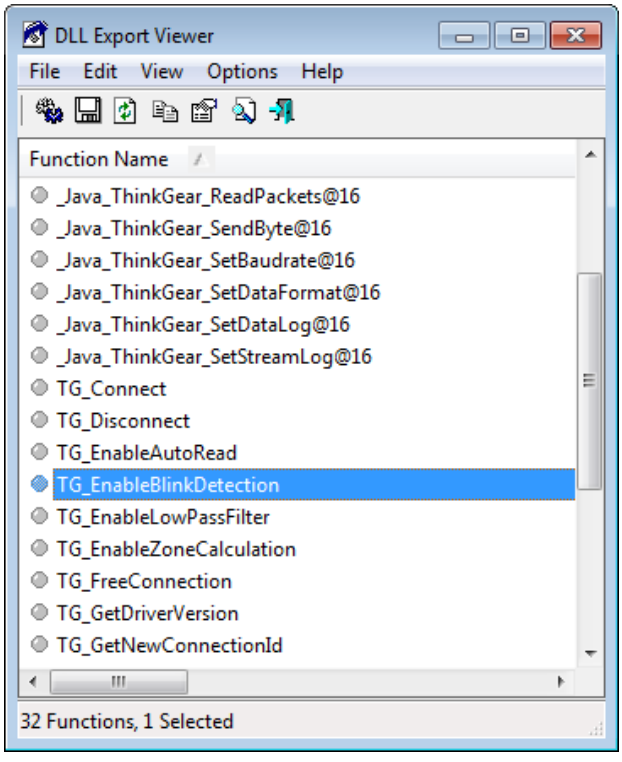


Figure 3. The Java MindWave Interface.

The ThinkGear DLL can be found in the \ThinkGear Communications Driver\win32\ subdirectory of the MindSet Development Tools, and comes with its own Java interface implemented using JNI (Java Native Interface) . However, it's lacking a method for switching on blink strength reporting (at least in the current 2.1 version), so I've replaced it with my own interface

It's easy to discover what functions are exported by the DLL with a tool such as DLL Export Viewer (a free download from http://www.nirsoft.net/utils/dll_export_viewer.html), which is shown in action in Figure 4.



 © Andrew Davison 2012

Figure 4. Some of the Functions Exported by the ThinkGear DLL.

Figure 4 lists some of the exported JNI native functions (e.g. Java_ThinkGear_ReadPackets) which wrap around corresponding C functions (e.g. TG_ReadPackets). Only function names are given in the export list, but there's a lot of information about the function parameters in the thinkgear.h header file that comes with the DLL.

The highlighted line in Figure 4 is the name of the C function TG_EnableBlinkDetection, which switches blink detection on and off (by default it's off). There's no JNI wrapper for this function, making NeuroSky's Java interface a little incomplete.

I decided to reimplement the Java interface using JNA since it doesn't require the re-compilation of the native code (in this case, the ThinkGear DLL, whose source isn't available anyway). The JNA library can be downloaded from https://github.com/twall/jna.

1.1. A ThinkGear Interface Using JNA

My ThinkGear.java JNA interface has three main parts: the Java equivalents of the library's constants and function prototypes, and an initial invocation of the DLL.

The constants and method signatures are based on information in the thinkgear.h header file in the \ThinkGear Communications Driver\win32\ subdirectory. One surprise of this header file is that it doesn't contain a prototype for TG_EnableZoneCalculation(), a function exported by the ThinkGear DLL. I've no idea what it does, so I based its parameters format on the other, documented 'enable' functions. The resulting ThinkGear.java code:

```
import com.sun.jna.Library;
import com.sun.jna.Native;


public interface ThinkGear extends Library
{
  // constants (based on those in thinkgear.h)

  /* max number of connections that can be requested
     before being required to free one */
  public static final int MAX_CONNECTION_HANDLES = 128;

  // baud rates
  public static final int BAUD_1200 = 1200;
  public static final int BAUD_2400 = 2400;
  public static final int BAUD_4800 = 4800;
  public static final int BAUD_9600 = 9600;
  public static final int BAUD_57600 = 57600;
  public static final int BAUD_115200 = 115200;

  // data formats
  public static final int STREAM_PACKETS = 0;
  public static final int STREAM_5VRAW = 1;
  public static final int STREAM_FILE_PACKETS = 2;
```

 © Andrew Davison 2012

```
  // create a run-time link to the DLL
  ThinkGear INSTANCE = (ThinkGear)
        Native.loadLibrary("ThinkGear", ThinkGear.class);


  // method prototypes (based on those in thinkgear.h)

  public int TG_GetDriverVersion();
  public int TG_GetNewConnectionId();

  public int TG_SetStreamLog(int connID, String filename);
  public int TG_SetDataLog(int connID, String filename);

  public int TG_WriteStreamLog(int connID,
                          int insertTimestamp, String msg);
  public int TG_WriteDataLog(int connID,
                          int insertTimestamp, String msg);

  public int TG_Connect(int connID, String serialPortName,
                int serialBaudrate, int serialDataFormat);

  public int TG_ReadPackets(int connID, int numPackets);

  public double TG_GetValue(int connID, int dataType);
  public int TG_GetValueStatus(int connID, int dataType);

  public int TG_SendByte(int connID, int b);

  public int TG_SetBaudrate(int connID, int serialBaudrate);
  public int TG_SetDataFormat(int connID,
                                      int serialDataFormat);

  public void TG_Disconnect(int connID);
  public void TG_FreeConnection(int connID);


  // methods not in the NeuroSky supplied Java interface
  public int TG_EnableLowPassFilter(int connID,
                                       int rawSamplingRate);

  public int TG_EnableBlinkDetection(int connID, int enable);
                  // enable uses 0 or 1 to act as a boolean

  public int TG_EnableAutoRead(int connID, int enable);

  public int TG_EnableZoneCalculation(int connID, int enable);
  // not mentioned in thinkgear.h but exported by the DLL;
  // I'm guessing at its arguments

}  // end of ThinkGear interface
```

The translation from C constants and prototypes to Java is straightforward since the majority of the C functions only use primitive types (e.g. int, float) without structs or pointers. For example, the TG_Connect() signature:

```
THINKGEAR_API int TG_Connect(int connectionId,
      const char *serialPortName, int serialBaudrate,
```

 © Andrew Davison 2012

```
      int serialDataFormat);
```

is represented by the Java prototype:

```
public int TG_Connect(int connID, String serialPortName,
                int serialBaudrate, int serialDataFormat);
```

Note that JNA maps String to the char* type.

The most unusual line in ThinkGear.java is probably:

```
ThinkGear INSTANCE = (ThinkGear)
        Native.loadLibrary("ThinkGear", ThinkGear.class);
```

This creates a run-time link (called INSTANCE) to thinkGear.dll, which must be in the same directory as the Java interface.

1.2. Wave Constants as an Enumeration

The thinkgear.h header defines the brainwave types as a series of #defines. For instance:

```
#define TG_DATA_ATTENTION   2
#define TG_DATA_MEDITATION  3
```

I could have mapped these to a series of Java static final integers, as I did with the baud rates and data formats. Instead I use a Wave enum class, which allows me to iterate through the type names at run-time. This is useful when printing data in the ReadMindWave.java example in the next section.

```
public enum Wave
{
  Battery(0),PoorSignal(1),
  Attention(2), Meditation(3),
  Raw(4),
  Delta(5), Theta(6),
  lowAlpha(7),highAlpha(8),
  lowBeta(9), highBeta(10),
  lowGamma(11),highGamma(12),
  BlinkStrength(37);

  private int code;

  private Wave(int code)
  {  this.code = code;  }

  public int getCode()
  {  return this.code; }

}  // end of Wave enum
```

 © Andrew Davison 2012

2. Reporting Brainwaves

ReadMindWave is a text-based application that uses my ThinkGear Java interface and thinkgear.dll to treat the MindWave as a serial COM device delivering packets of data.

ReadMindWave reports all the EEG information it receives, including the attention and meditation values, the blink strength, all the EEG bands (i.e. the delta, theta, alpha, beta, and gamma), the raw EEG data packets (which arrive very frequently at 512Hz), signal quality, and the battery setting. This is a little excessive, so a slight variation of the application reports only the attention, meditation, and blink strength values.

The ReadMindWave() constructor sets up the MindWave connection, then enters a loop that iterates for about 60 seconds, reporting the MindWave brainwave values roughly every second.

```
// globals
private static final int COM_PORT = 4;
     // this value depends on your MindWave installation;
     // change as required


private static final int POLL_INTERVAL = 5;   // ms
        // how frequently to read the MindWave data

private static final int REPORT_INTERVAL = 1000;   // ms
        // how frequently to report the MindWave data to the user

private static final int MAX_RUN_TIME = 60000;   // ms

private ThinkGear tg = ThinkGear.INSTANCE;


public ReadMindWave()
{
  System.out.println("ThinkGear DLL version: " +
                                tg.TG_GetDriverVersion());

  int connID = connect(COM_PORT);
  tg.TG_EnableBlinkDetection(connID, 1);  // switch on

  logInput(connID);
  long startTime = System.currentTimeMillis();
  long reportTime = startTime;

  int totalPacks = 0;
  while ((System.currentTimeMillis() - startTime) < MAX_RUN_TIME) {
    totalPacks += readPackets(connID);
    if ((System.currentTimeMillis() - reportTime) > REPORT_INTERVAL){
      if (!hasRaw(connID))
        System.out.println("no data");
      else if (!hasSignal(connID))
        System.out.println("low signal");
      else
        printAllWaves(connID, totalPacks);
        // printWaves(connID);   // a less verbose report

      reportTime = System.currentTimeMillis();
    }
```

 © Andrew Davison 2012

```
    try {
      Thread.sleep(POLL_INTERVAL);
    }
    catch (InterruptedException e) {}
  }

  tg.TG_FreeConnection(connID);
}  // end of ReadMindWave()
```

The JNA link to the ThinkGear library is stored in the tg instance, which is used when calling the Java wrapper functions for the DLL. For example, a call to tg.TG_GetDriverVersion() is mapped to the C function TG_GetDriverVersion() which return the DLL's version as an integer (in my code, 21, representing version 2.1 of the TGCD).

The loop inside ReadMindWave() is a little complicated because the polling frequency for retrieving MindWave data is roughly every 5 ms, while the reporting interval for printing the brainwave values to standard output is about 1 second. The rapid polling rate means that no data packets are lost, and the less frequent report rate reduces the amount of data output.

2.1. Connecting to the MindWave

The steps involved in connecting to the MindWave are hidden inside the connect() method. A connection ID is obtained first, then it and the COM port number are employed to open the link.

```
private int connect(int comPort)
{
  int connID = tg.TG_GetNewConnectionId();
  if(connID < 0) {
    System.out.println("Could not get a connection ID");
    System.exit(1);
  }

  int connResult = tg.TG_Connect(connID, "\\\\.\\COM" + comPort,
                     ThinkGear.BAUD_9600, ThinkGear.STREAM_PACKETS);
  if(connResult < 0) {
    System.out.println("Could not connect to COM " + comPort);
    System.exit(1);
  }

  return connID;
}  // end of connect()
```

The COM port number is assigned to the MindWave when its software is first installed, and can be checked by calling the MindWave Manager tool in the NeuroSky utilities menu. It reports both the MindWave ID and COM Port, as in Figure 5.

 © Andrew Davison 2012

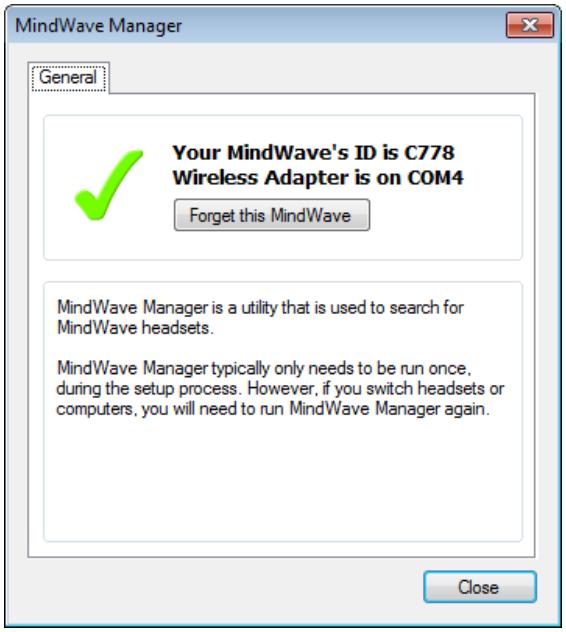


Figure 5. The MindWave Manager.

The other arguments to TG_Connect() are the baud rate and communication protocol.

The connection ID is returned by connect() since its required in all subsequent MindWave function calls.

2.2. Reading Data Packets

The heart of my readPackets() method is a call to the ThinkGear TG_ReadPackets() function, which reads in as many packets as are currently available. TGCD automatically uses this information to update the brainwave values for attention, meditation, and the other types. When a brainwave value is updated, a corresponding status flag for the wave is set to true.

```
// globals
private boolean hasBlinked = false;
     /* this flag is used to remember a blink strength update
        until the value is printed out */


private int readPackets(int connID)
{
  int numPackets = tg.TG_ReadPackets(connID, -1);
                          // read all available packets
  if (numPackets == -1) {
    System.out.println("Invalid connection ID");
    System.exit(1);
  }
  else if (numPackets == -2)
    return 0;
  else if (numPackets == -3) {
    System.out.println("Serial stream read error");
    System.exit(1);
  }
```

 © Andrew Davison 2012

```
  if (isUpdated(connID, Wave.BlinkStrength))
    hasBlinked = true;

  return numPackets;
}  // end of readPackets()
```

TG_ReadPackets() can fail in a number of ways which are detected by looking at its integer result.

isUpdated() accesses the ThinkGear status integer for a specified wave:

```
private boolean isUpdated(int connID, Wave wave)
{  return tg.TG_GetValueStatus(connID, wave.getCode()) != 0;  }
```

TG_GetValueStatus() returns a non-zero integer if the wave was updated by the most recent call to TG_ReadPackets(), and 0 otherwise.

Back in readPackets(), if the blink strength value was updated then the global boolean, hasBlinked, is set to true. This may seem pointless when the wave value's status can be determined at any time by calling TG_GetValueStatus(). But there's a subtle issue with relying on wave status integers – they're all reset to 0 when TG_ReadPackets() is next called, and only assigned a non-zero if that particular call contains data for that wave.

A problem arises because my code reports brainwave values much less frequently than it calls TG_ReadPackets(), so that by the time a wave is examined by my code, its ThinkGear status integer may have been reset to 0 by a new call to TG_ReadPackets().

This is most apparent with the blink strength type since it's updated infrequently (only when the user blinks their eye). When the Wave.BlinkStrength value is changed because of a blink, its status integer is set to non-zero. However, there will probably be multiple further calls to TG_ReadPackets() before the blink value is examined by my code. By that time, it's almost certain that the Wave.BlinkStrength status integer will have been reset to 0.

The only way to be certain that the blink strength has changed since its last access by my code is to use a separate hasBlinked flag. I use this technique in the printAllWaves() reporting method described at the end of section 2.4.

2.3. Checking the Data

My methods hasRaw() and hasSignal() illustrate two simple ways that the packet data can be examined for problems. hasRaw() retrieves the ThinkGear raw wave value, which will only be 0 if no packet data was found by the last call TG_ReadPackets():

```
private boolean hasRaw(int connID)
// is there some raw data available?
{  return (get(connID, Wave.Raw) != 0);  }


private double get(int connID, Wave wave)
// return the value for this wave type
```

 © Andrew Davison 2012

```
{  return tg.TG_GetValue(connID, wave.getCode());  }
```

hasSignal() retrieves the Wave.PoorSignal value, which ideally should be 0 indicating a perfect wireless connection to the MindWave. Higher values mean the reception is weaker, which will impact the packet data quality. A Wave.PoorSignal value of 200 means that no signal was detected, probably because the MindWave isn't being wore by the user:

```
// global
private static final int STRONG_SIGNAL = 50;
         // smaller is stronger, with 0 the most powerful


private boolean hasSignal(int connID)
// is the signal strong enough?
{
  double signal = get(connID, Wave.PoorSignal);
  if (signal == 200) {   // special signal value
    System.out.println("ThinkGear contacts not touching the skin");
    return false;
  }
  return (signal < STRONG_SIGNAL);
}  // end of hasSignal()
```

hasSignal() assumes a 'strong' signal is one with a value below 50, a number I arrived at by experimenting with the MindWave.

It's useful to have these extra tests since the MindWave is rather temperamental, often taking several seconds to start sending data packets, and very ready to transmit a poor signal strength value if the forehead electrode or earlobe clip are a tiny bit out of place. Sometimes it's necessary to switch the headset on and off a few times before it can be coaxed into sending any data at all.

2.4. Reporting the Data

My printAllWaves() method iterates through the Wave enum types, printing their current values and update status:

```
private void printAllWaves(int connID, int totalPacks)
{
  System.out.println("---- Report (" + totalPacks + ") ----");
  for (Wave wave : Wave.values())
    System.out.println("  " + wave + ": " + get(connID, wave) +
                       " [" + isUpdated(connID, wave) + "]");
}  // end of printAllWaves()
```

A typical fragment of the output is:

```
---- Report (744) ----
  Battery: 0.0 [false]
  PoorSignal: 26.0 [false]
  Attention: 0.0 [false]
```

 © Andrew Davison 2012

```
  Meditation: 0.0 [false]
  Raw: -972.0 [true]
  Delta: 107138.0 [false]
  Theta: 32954.0 [false]
  lowAlpha: 15222.0 [false]
  highAlpha: 112417.0 [false]
  lowBeta: 20124.0 [false]
  highBeta: 31006.0 [false]
  lowGamma: 58019.0 [false]
  highGamma: 80917.0 [false]
  BlinkStrength: 0.0 [false]

---- Report (1260) ----
  Battery: 0.0 [false]
  PoorSignal: 26.0 [false]
  Attention: 0.0 [false]
  Meditation: 0.0 [false]
  Raw: 928.0 [true]
  Delta: 635747.0 [false]
  Theta: 2651185.0 [false]
  lowAlpha: 5760.0 [false]
  highAlpha: 522243.0 [false]
  lowBeta: 232968.0 [false]
  highBeta: 268824.0 [false]
  lowGamma: 128748.0 [false]
  highGamma: 1108054.0 [false]
  BlinkStrength: 0.0 [false]

---- Report (1776) ----
      :
```

Almost all the waves report false for their update status even though their values are changing from one report to the next. The reason is that their ThinkGear status integers are being reset to 0 by TG_ReadPackets() before printAllWaves() has a chance to access their values. One way to avoid this problem is to introduce global booleans for all the Wave types, similar to hasBlinked for the blink strength.

Another aspect of the ThinkGear's library is that if new data isn't received for a wave type then the corresponding wave value retains its current value; it is not set back to 0. This means that printing a series of high values for a wave type, such as the attention level, does not necessarily mean that such data is being received for that wave. If the sequence of values repeats the same number, then it probably means that the wave hasn't been updated recently. Unfortunately, this suspicion can't be checked by looking at the wave's status integer for the reasons given above.

In practice, the most useful wave types are attention, meditation, and blink strength, which are reported using the following alternative method:

```
// globals
private boolean hasBlinked = false;


private void printWaves(int connID)
{
  System.out.println("---- Report ----");
  System.out.println("  Attention: " +
                       get(connID, Wave.Attention));
  System.out.println("  Meditation: " +
```

 © Andrew Davison 2012

```
                          get(connID, Wave.Meditation));
  if (hasBlinked) {
    System.out.println("  Blink Strength: " +
                          get(connID, Wave.BlinkStrength));
    hasBlinked = false;    // reset flag after the value is printed
  }
}  // end of printWaves()
```

printWaves() shows how the hasBlinked global flag is utilized to detect if the blink strength has changed since the last call to printWaves(). Only after the strength is printed is the flag set to false. Also, it can only be toggled back to true by readPackets() when it receives a new blink strength. A similar coding style could be employed for the other wave types to ensure that their updates could be accurately detected.

2.5. Logging Data Packets

The ThinkGear library has two logging functions which cause incoming data packets to be copied to text-based log files. Both are called from my logInput() method before the read-packets loop is entered:

```
// globals
// files for logging the MindWave data
private static final String STREAM_LOG_FNM = "bytesLog.txt";
private static final String PACKETS_LOG_FNM = "packetsLog.txt";


private void logInput(int connID)
{
  // record input bytes to a log file
  int resultCode = tg.TG_SetStreamLog(connID, STREAM_LOG_FNM);
  if(resultCode == 0)
    System.out.println("Recording input bytes to " + STREAM_LOG_FNM);
  else
    System.out.println("Could not record input bytes to " +
                                                 STREAM_LOG_FNM);

  // record input packets to a log file
  resultCode = tg.TG_SetDataLog(connID, PACKETS_LOG_FNM);
  if(resultCode == 0)
    System.out.println("Recording input packets to " +
                                          PACKETS_LOG_FNM);
    else
      System.out.println("Could not record input packets to " +
                                          PACKETS_LOG_FNM);
  }  // end of logInput()
```

TG_SetStreamLog() requests that the packet data be logged as time-stamped sequences of hexadecimal strings. A typical line from the log:

```
1351762778.281:  02 FD 7F 01 AA AA 04 80 02 FF ED 91 AA AA 04 80 02
02 74 07 AA AA 04 80 02 04 18 61 AA AA 04 80 02 04 5C 1D
```

 © Andrew Davison 2012

A line begins with a time-stamp in seconds since the UNIX epoch time (midnight, 1st January 1970),with a millisecond fractional part. After the ":" there will often be multiple packets, some spread over more than one line. For instance, the line above starts with four hexadecimals that finish a packet that started on the previous line.

A packet starts with two "AA" hexadecimals (called SYNC codes) followed by a packet length hexadecimal, and a checksum hexadecimal at the end. In between there will be information about different brainwave types, distinguished by hex codes. I'll talk about these codes when I explain my log reader application, ReadPacketsLog.java, in section 4.

Most of my information about the log format comes from the MindSet Development Tools (MDT) download, in its \ThinkGear\mindset_communications_protocol.pdf document, which explains the serial data layout for the MindWave. This isn't quite the same as the log format, but is close.

My ReadPacketsLog application (described below) reads the other type of log text file, which is created by a call to TG_SetDataLog(). It utilizes a higher-level format, where each log line contains the data for a single brainwave type; as a consequence, it's much easier to read.

There's no need for my logInput() method to create both types of log file. I included both only to show how the logging functions are called.

3. Visualizing the MindWave Data

The textual output of even a limited amount of brainwave data (i.e. attention and meditation levels and blink strength) is still too much to follow easily. The natural next step is to display it in a dynamically updating chart, as in Figure 2, and again in Figure 6.

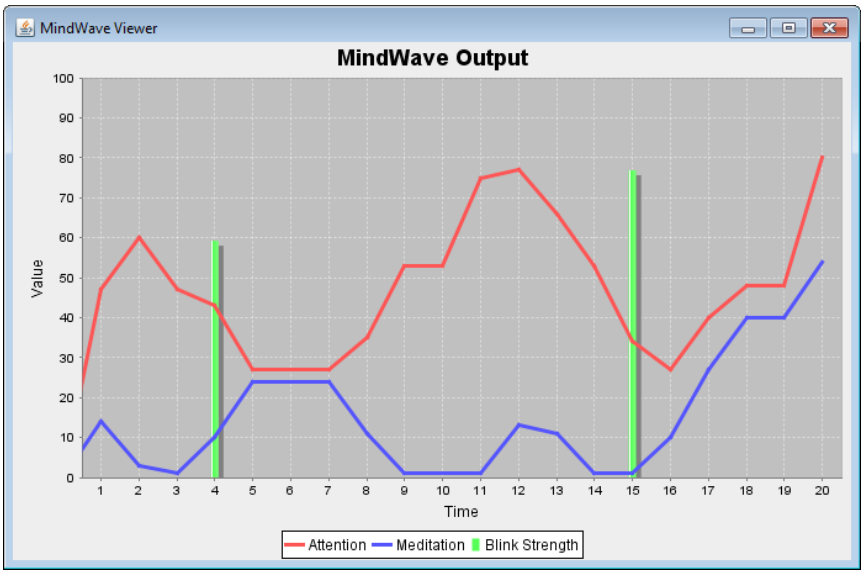


Figure 6. Another Example of MindWave Charting.

 © Andrew Davison 2012

The attention and meditation levels are drawn as red and blue line plots which are updated roughly every second. The blink strength data is a green bar chart, with a bar drawn only when a blink is detected.

The chart's x-axis is in seconds, and scrolls to the left as time progresses. For example, Figure 2 shows the brainwave output after about 100 seconds has passed.

The chart's y-axis goes from 0 to 100 to cover the data range of the attention and meditation levels. Blink strength data can range between 0 and 255, but is scaled to span 0 to 100 so it can use the same axes as the line plots.

The attention and meditation graphs are based on the current levels returned by the ThinkGear library, which retain their old values if new values aren't detected when brainwave packets are read. This means that if a line runs horizontally for several seconds, as in Figure 6 between 5 and 8 seconds, it most likely means that no new data was received, not that the input is running at a constant level.

Another aspect of chart ploting is that when the MindWave's wireless signal is too low, or no raw data is being received, then the chart stops being updated. Perhaps it would have been better to set the levels to 0 in those cases, but the lack of new data points in the graph makes it clear that there's a problem with the MindWave.

The UML class diagrams for this application, which is called MindWaver, are shown in Figure 7.

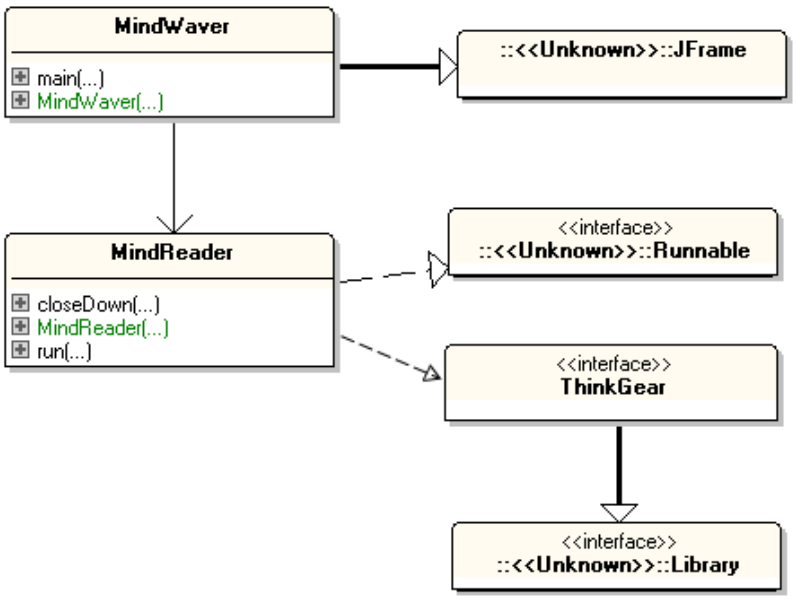


Figure 7. The MindWaver Class Diagrams.

MindWaver is a JFrame which manages the initialization of the chart and adds it to a panel. I use the JFreeChart library (http://www.jfree.org/jfreechart/) to create a combination line and bar chart – two line plots for the attention and meditation levels and bars for the blink strength. The graph data is stored in three JFreeChart XYSeries instances, whose references are passed to the MindReader object. MindReader deals with reading data from the MindWave and updating the series.

 © Andrew Davison 2012

MindReader is similar to my earlier ReadMindWave class, only differing in the way that the read-packets loop is controlled. MindReader is threaded so that the loop, which may execute for a long time, doesn't affect the GUI, and especially chart redrawing.

3.1. Initializing the Application

MindWaver initializes three XYSeries objects for holding plot data, builds a combined line and bar chart, and adds it to a panel in the application's window.

```
// globals
private static final int X_RANGE = 20;
   // a spread of 20 seconds are visible on the JFreeChart graph

private XYSeries attnSeries, mediSeries, blickSeries;
   // JFreeChart data sets for Attention, Meditation, and
   // Blink Strength

private MindReader mwReader;


public MindWaver()
{
  super("MindWave Viewer");

  // initialize 3 series (for 2 line and 1 bar charts)

  attnSeries = new XYSeries("Attention");   // 2 line charts
  mediSeries = new XYSeries("Meditation");
  XYSeriesCollection linesData = new XYSeriesCollection();
  linesData.addSeries(attnSeries);
  linesData.addSeries(mediSeries);

  blickSeries = new XYSeries("Blink Strength");   // a bar chart
  blickSeries.add(X_RANGE, null);
          // hack to start dynamic addition at left edge of graph
  XYSeriesCollection barData = new XYSeriesCollection();
  barData.addSeries(blickSeries);

  JFreeChart chart = createCombinedChart(linesData, barData);
  ChartPanel chartPanel = new ChartPanel(chart);
  Container c = getContentPane();
  c.add(chartPanel);

  mwReader = new MindReader(attnSeries, mediSeries, blickSeries);
    /* start reading from the MindWave, updating the three chart
       data series */

  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    {  mwReader.closeDown(); }    // close MindWave connection
    });

  pack();
  setLocationRelativeTo(null);  // center the window
  setVisible(true);
}  // end of MindWaver()
```

 © Andrew Davison 2012

One hacky part of the method is the addition of a null data point to the blink series at the 20 second mark:

```
blickSeries.add(X_RANGE, null);    // X_RANGE is 20
```

This forces the chart to be rendered so that the 20 second x-coordinate is visible. In addition, createCombinedChart() sets the chart's visible x-axis range to be 20 seconds wide. These two things mean that the left edge of the chart will start at 0 seconds. If I didn't add the null value, then the 0 seconds tick mark would start near the right edge of the chart, with the x-coordinate for -19 seconds on the left edge.

MindReader's packet-reading loop continues executing until MindWaver calls MindReader.closeDown() when its close-box is clicked.

MindWaver creates three XYSeries objects for the attention and meditation levels and blink strength. The first two are grouped into a single XYSeriesCollection since they will be rendered in the same way, as connected lines. The blink strength series is placed in its own XYSeriesCollection because it will be drawn as a bar chart. The two collections are passed to createCombinedChart() which sets up their rendering features.

3.2. Creating the Chart

createCombinedChart() initializes the look of the chart – a combination of two line plots and a bar chart.

```
// global
private static final int X_RANGE = 20;
   // a spread of 20 seconds are visible on the JFreeChart graph


private JFreeChart createCombinedChart(
                            XYSeriesCollection linesData,
                            XYSeriesCollection barData)
{ // create xy line plot renderer
  XYItemRenderer linesRenderer = new StandardXYItemRenderer();
  linesRenderer.setBaseToolTipGenerator(
      new StandardXYToolTipGenerator(
          StandardXYToolTipGenerator.DEFAULT_TOOL_TIP_FORMAT,
          new DecimalFormat("0.00"), new DecimalFormat("0.00")
      )
  );
  // thicker lines for both data series
  linesRenderer.setSeriesStroke(0, new BasicStroke(3.0f));
  linesRenderer.setSeriesStroke(1, new BasicStroke(3.0f));

  // create axes for the plot (same for all data)
  NumberAxis domainAxis = new NumberAxis("Time");
  domainAxis.setStandardTickUnits(
                     NumberAxis.createIntegerTickUnits());
  domainAxis.setAutoRange(true);
  domainAxis.setFixedAutoRange(X_RANGE);  // seconds

  NumberAxis rangeAxis = new NumberAxis("Value");
  rangeAxis.setRange(0.0, 100.0);
```

 © Andrew Davison 2012

```
  // start building the plot with the lines data
  XYPlot plot = new XYPlot(linesData, domainAxis, rangeAxis,
                                                    linesRenderer);
  // customize the plot
  plot.setBackgroundPaint(Color.lightGray);
  plot.setDomainGridlinePaint(Color.white);
  plot.setRangeGridlinePaint(Color.white);

  // create bar chart renderer
  XYItemRenderer barRenderer = new XYBarRenderer(0.8);   // thin bars
  barRenderer.setBaseToolTipGenerator(
      new StandardXYToolTipGenerator(
          StandardXYToolTipGenerator.DEFAULT_TOOL_TIP_FORMAT,
          new DecimalFormat("0.00"), new DecimalFormat("0.00")
      )
  );

  // add bar chart data and renderer to the plot
  plot.setDataset(1, barData);
  plot.setRenderer(1, barRenderer);

  // return a chart containing the overlaid plot...
  return new JFreeChart("MindWave Output",
                JFreeChart.DEFAULT_TITLE_FONT, plot, true);
}  // end of createCombinedChart()
```

The attention and meditation levels data are stored in an XYSeriesCollection object called linesData, and the blink strength data in the barData XYSeriesCollection object. The line rendering of linesData is managed by a StandardXYItemRenderer object, while the bars are handled by an XYBarRenderer instance.

All the data sets and renderers are packaged together into a single XYPlot object so the three graphs appear in a single chart.

3.3. Connecting to the MindWave

The same connect() method as in the ReadMindWave application is called by the MindReader constructor:

```
// globals
private static final int COM_PORT = 4;

private ThinkGear tg = ThinkGear.INSTANCE;
private int connID;
private XYSeries attnSeries, mediSeries, blickSeries;
    // data sets for attention, meditation, and blink strength


public MindReader(XYSeries aSeries, XYSeries mSeries,
                                          XYSeries bSeries)
{ attnSeries = aSeries;
  mediSeries = mSeries;
  blickSeries = bSeries;

  connID = connect(COM_PORT);
  tg.TG_EnableBlinkDetection(connID, 1);

  new Thread(this).start();   // start polling the MindWave
```

 © Andrew Davison 2012

```
} // end of MindReader()
```

A thread is started at the end of the constructor, so its long-lived packet reading loop won't impact the rest of the application.

3.4. Reading MindWave Packets

The loop inside MindReader's run() method is essentially the same as the loop in ReadMindWave, except that it terminates when a global boolean, isRunning, is set to false by MindReader.closeDown().

```
// globals
private static final int POLL_INTERVAL = 5;   // ms
        // how frequently to read the MindWave data

private static final long REPORT_INTERVAL = 1000;   // ms
        // how frequently to report the MindWave data to the user

private ThinkGear tg = ThinkGear.INSTANCE;
private int connID;
private volatile boolean isRunning = false;


public void run()
{
  isRunning = true;
  long reportTime = System.currentTimeMillis();
  int reportCount = 0;

  while (isRunning) {
    readPackets(connID);
    if ((System.currentTimeMillis()-reportTime) > REPORT_INTERVAL) {
      if (!hasRaw(connID))
        System.out.println("no data");
      else if (!hasSignal(connID))
        System.out.println("low signal");
      else {
        reportWaves(connID, reportCount);
        reportCount++;
      }
      reportTime = System.currentTimeMillis();
    }
    try {
      Thread.sleep(POLL_INTERVAL);
    }
    catch (InterruptedException e) {}
  }

  // close down
  tg.TG_FreeConnection(connID);
  System.exit(1);
}  // end of run()


public void closeDown()
// called from MindWaver when its close-box is clicked
{  isRunning = false;  }
```

 © Andrew Davison 2012

run() calls the methods readPackets(), hasRaw(), and hasSignal(), which are identical to the ones in ReadMindWave.

When the read-loop terminates, the connection to the MindWave is closed by a call to the ThinkGear library function TG_FreeConnection().

3.5. Reporting the Wave Data

reportWaves() prints the attention, meditation and blink strength values to standard output just as before. The important change is that the method also adds data points to the waves' XYSeries. JFreeChart automatically detects those additions, and redraws the chart.

```
// globals
private boolean hasBlinked = false;
     /* this flag is used to remember a blink strength update
        until the value is printed out */

private XYSeries attnSeries, mediSeries, blickSeries;


private void reportWaves(int connID, int reportCount)
{
  System.out.println("---- Report (" + reportCount + ")----");

  double attention = get(connID, Wave.Attention);
  System.out.println("  Attention: " + attention);
  attnSeries.add(reportCount, attention);

  double meditation = get(connID, Wave.Meditation);
  System.out.println("  Meditation: " + meditation);
  mediSeries.add(reportCount, meditation);

  if (hasBlinked) {
    double bs = get(connID, Wave.BlinkStrength);
    System.out.println("  Blink Strength: " +  bs);
    blickSeries.add(reportCount, bs/2.55);   // scale 0-255 ==> 0-100
    hasBlinked = false;     // reset flag after the value is printed
  }
  else     // don't keep showing the old value
    blickSeries.add(reportCount, 0);
}  // end of reportWaves()
```

A global hasBlinked flag is utilized, as in ReadMindWave. It's set to true in readPackets() when the blink strength is updated, and only turned off when the data point has been added to the chart. When hasBlinked is false, a zero height bar is added to the chart's data meaning that a blink is only rendered as a single bar on the chart .

Since blink strengths can range up to 255, it's necessary to scale them down to between 0 and 100 so they'll fit into the y-axis range of the levels.

 © Andrew Davison 2012

4. Displaying Log Data

ReadMindWave writes incoming brainwave packets into both bytes-based and packets-based log files. This section describes an application called ReadPacketsLog which prints the contents of a packets-based log file in a high-level, nicely formatted manner.

Typical lines in a packets-based log file look like the following:

```
1351824599.740: [80]    236, 00EC, -1.615836
1351824599.740: [80]    456, 01C8, -0.325513
1351824599.747: [02] 0
1351824599.747: [83]   70552, 0x00011398,   68964, 0x00010D64,
2170, 0x0000087A,   72481, 0x00011B21,     644, 0x00000284,   54575,
0x0000D52F,     768, 0x00000300,  251315, 0x0003D5B3,
1351824599.747: [04] 75
1351824599.747: [05] 61
1351824599.747: [80]    544, 0220,  0.190616
```

Each line represents the data for a single brainwave type, prefixed by a timestamp before the ":". The timestamp is in seconds since the UNIX epoch time, along with milliseconds after the decimal point.

The hexadecimal number between the square brackets is a code representing the different types of packets, as explained in the \ThinkGear\mindset_communications_protocol.pdf document in the MindSet Development Tools (MDT). For instance, 0x80 is a for a packet containing raw data, 0x02 is a poor signal packet. 0x83 is for a collection of eight values for the delta, theta, low-alpha, high-alpha, low-beta, high-beta, low-gamma, and mid-gamma brainwave types. Of particular importance are the codes 0x04 (attention level), 0x05 (meditation level), and 0x16 (blink strength).

I was unable to locate any information on the format of the numerical data following a code, although it's fairly easy to guess by comparing it with the data returned by calls to the ThinkGear TG_GetValue() function.

ReadPacketsLog can be called in two ways – either with an option set to show raw data, which produces voluminous output, or in default mode which doesn't report raw information. Using the latter, the log file fragment shown above is displayed as:

```
[1485 ms] EEG
  delta: 70552;  theta: 68964
  alpha (low): 2170 -- (high): 72481
  beta (low): 644 -- (high): 54575
  gamma (low): 768 -- (high): 251315
[1485 ms] Attention: 75
[1485 ms] Meditation: 61
```

The UNIX epoch time is replaced by the elapsed time in milliseconds since the creation of the log, and named brainwave types are associated with their decimal values.

 © Andrew Davison 2012

Processing a Log Line

The top-level of ReadPacketsLog is a loop that reads in a line at a time from the log file, passing the string to a processLine() method. It splits the line into two parts around the ":" – the UNIX time before, and the brainwave code and data after:

```
private static void processLine(String line)
{
  String[] lineData = line.split(":");
  if (lineData.length == 2) {
    long eTime = elapsedTime(lineData[0]);
    printPacket(lineData[1].trim(), eTime);
  }
  else
    System.out.println("Line format incorrect: \"" + line + "\"");
}  // end of processLine()
```

elapsedTime() stores the first timestamp (i.e. the one pulled from the first log line) as a global millisecond integer, called startTime. Subsequent calls to elapsedTime() subtract this global from the current timestamp to get the elapsed time in milliseconds.

```
// global
private static long startTime = 0;


private static long elapsedTime(String timeStr)
{
  long eTime = 0;
  try {
    long time = Long.parseLong(timeStr.replace(".", ""));
    // remove the "."  in text such as "1350614425.491",
    // and convert to long
    if (startTime == 0)
      startTime = time;
    eTime = time-startTime;
  }
  catch (NumberFormatException e)
  { System.out.println("Could not parse time: \""+timeStr+"\"");  }
  return eTime;
}  // end of elapsedTime()
```

printPacket() is passed a string that starts with a 2-digit hexadecimal code in square brackets. For example:

```
      [80]    155, 009B, -2.090909
or    [83] 1887198, 0x001CCBDE,  ...,  669251, 0x000A3643,
or    [04] 40
```

Most of the codes are listed in the CODE definitions table in the TGC developer guide in the \ThinkGear\mindset_communications_protocol.pdf document in the MindSet Development Tools.

I'm not entirely sure about the purpose of all the digits that follow a code, but the first value always seems to be the brainwave decimal value, except in the case of code 83 which is followed by data for eight EEG waves.

 © Andrew Davison 2012

printPacket() is essentially a multi-way branch based on the code value:

```
// globals
private static boolean showRaws = false;
    // if this is set to true then lots of raw data is printed


private static void printPacket(String dataStr, long eTime)
{
  String codeStr = dataStr.substring(1, 3);

  String[] params = dataStr.substring(4).trim().split(",");
  for(int i=0; i < params.length; i++)
    params[i] = params[i].trim();

  if (codeStr.equals("02")) {
    if (!params[0].equals("0"))    // don't print normal signal data
      System.out.println("\n["+eTime+" ms] Poor Signal: "+params[0]);
  }
  else if (codeStr.equals("04"))
    System.out.println("\n[" + eTime + " ms] Attention: "+params[0]);
  else if (codeStr.equals("05"))
    System.out.println("\n[" + eTime +" ms] Meditation: "+params[0]);
  else if (codeStr.equals("16"))
    System.out.println("\n[" + eTime + " ms] Blink Strength: " +
                                                        params[0]);
  else if (codeStr.equals("80")) {
    if (showRaws)    // a flag that help reduce the amount of output
      System.out.println("\n[" + eTime + " ms] Raw: " + params[0]);
  }
  else if (codeStr.equals("83")) {
    System.out.println("\n[" + eTime + " ms] EEG");
    printEEGs(params);
  }
  else
    System.out.println("\n["+eTime+" ms] Unknown code: " + codeStr);
}  // end of printPacket()
```

The code 83 data consists of eight different wave values, given in both decimal and hexadecimal form. For example:

```
299209, 0x000490C9, 193287, 0x0002F307, 22743, 0x000058D7, 11035,
0x00002B1B, 21571, 0x00005443, 3502, 0x00000DAE, 3820, 0x00000EEC,
71882, 0x000118CA,
```

They appear to be for the waves: delta, theta, low alpha high alpha, low beta, high beta, low gamma, and high gamma. However, the alpha and beta ranges seem to be swapped, but I've left them in that order. printEEGs() prints the decimal value of each wave:

```
private static void printEEGs(String[] params)
{
{ System.out.println("  delta: " + params[0] +
                     ";  theta: " + params[2]);
  System.out.println("  alpha (low): " + params[4] +
                       " -- (high): " + params[6]);
  System.out.println("  beta (low): " + params[8] +
                      " -- (high): " + params[10]);
  System.out.println("  gamma (low): " + params[12] +
```

 © Andrew Davison 2012

```
                          " -- (high): " + params[14]);
}  // end of printEEGs()
```

 © Andrew Davison 2012